

ORCA/M
A Macro Assembler
for the
Apple IIgs

Mike Westerfield
Phil Montoya

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

Copyright 1984 - 1991
By The Byte Works, Inc.
All Rights Reserved

Copyright 1986 - 1989
Apple Computer, Inc.
All Rights Reserved

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

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

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

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

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

Use of Libraries - The inclosed 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 include the libraries of 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 the licensing, and the document must give credit for using libraries from ORCA/M. For details, please contact the Byte Works, Inc.

ORCA/M is a trademark of the Byte Works, Inc.
Apple is a registered trademark of Apple Computer, Inc.

Program, Documentation and Design
Copyright 1984 - 1991
The Byte Works, Inc.

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

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

Portions of this documentation are derived from material copyrighted by Apple Computer, Inc. These portions are used with permission of Apple Computer.

Apple is a registered trademark of Apple Computer, Inc.

Table of Contents

Welcome To ORCA	xxiii
Express ORCA Start Up	xxiii
About This Manual	xxiii
Visual Cues	xxiv
What You Need	xxiv
Additional Reading And Reference	xxiv
Reading List	xxv
 Chapter 1 – Getting Started	 1
The ORCA Shell	1
Shell Commands	1
The Editor	2
Calling The Editor	2
Assembling and Linking a Program	3
 Chapter 2 – The Shell	 7
The Shell Commands	7
Listing a Directory	8
The Shell Line Editor	9
Entering Commands	9
Scrolling Through Commands	10
Using Wildcards	11
Required and Optional Parameters	12
Types of Commands	12
 Chapter 3 – The Editor	 13
Starting The Editor	13
Help	13
Editor Modes	14
Entering Text	14
Moving Through A File	14
Small Movements	14
Medium Movement	15
Large Movements	15
Modifying Text	16
Deleting Text	16
Inserting Text	17
Advanced Features	17
Search / Search With Replace	17
Search	17
Search With Replacement	19
Cut, Copy, Paste and Clear	19
Cut	20
Copy	20

Table of Contents

Paste	20
Clear	21
Closing The File	21
For More Information...	21
Chapter 4 – The Fundamental Assembler Directives	23
Introduction	23
The Assembly Language Statement	23
Label	24
Op code	25
Operand	25
Comment	25
Comment Lines	26
Directives	26
KEEP	26
START and END	26
Equates	27
ANOP	28
DC and DS	28
Global Labels	30
DATA Segments	31
Chapter 5 – Advanced Shell Features	35
File Naming Conventions	35
What Is A File?	35
File Names	35
Path Names	36
Directory Walking	36
Device Names	37
Standard Prefixes	37
Redirecting Input and Output	40
The Program Development Process	40
Assembling A Program	41
Linking A Program	42
Executing a Program	43
Chapter 6 – Advanced Assembler Directives	45
COPY and APPEND	45
Format Control Directives	46
Setting Case Sensitivity	47
Setting The Most Significant Bit In Characters	47
Load Segments	48
Changing The Word Size	50
Miscellaneous Directives	50
MERR Directive	50
Using the 65C02 and 6502	50
Positioning Code And The ORG Directive	51
Chapter 7 – Using Macros	53

Table of Contents

Tools of the Trade	53
The Macro Library	56
More About Macro Parameters	57
Chapter 8 – Writing Macros	59
MCOPY, MACRO and MEND	59
Basic Parameter Passing	60
Defining Symbolic Parameters	62
Changing and Using Symbolic Parameters	63
String Manipulation	64
Conditional Assembly Branches	65
Attributes	65
Chapter 9 – Writing Programs and Utilities	69
Shell Text Programs	69
Memory Management	72
Installing a New Utility	72
Installing a Compiler or Editor	73
System Programs	73
Other Kinds of Programs	74
Classic Desk Accessories	75
New Desk Accessories	75
Initialization Programs	75
Control Panel Devices	75
Chapter 10 – Programming the Shell	77
What Is an EXEC File?	77
Variables	78
The SET Command	79
The UNSET Command	80
The Classic Start: Hello, World	80
Passing Parameters	81
Variables Defined by the Shell	82
Variable Scope	83
Loops	84
Conditional Execution	86
Chapter 11 – Introduction To The Shell	89
What is the Shell?	89
The Command Processor	89
The Text Editor	90
The Assembly Process	90
Assembly Language Programs	91
Chapter 12 – The Command Processor	93
The Line Editor	93
Command Name Expansion	94
Editing A Command On The Command Line	94
Multiple Commands	95

Table of Contents

Scrolling Through Commands	95
Command Types	95
Built-in Commands	96
Utilities	96
Language Names	96
Program Names	97
Standard Prefixes	97
Prefixes 0 to 7	99
File Names	100
Wildcards	101
Types of Text Files	102
EXEC Files	102
Passing Parameters Into EXEC Files	103
Programming EXEC Files	103
Variables	103
Logical Operators	106
Entering Comments	107
Redirecting Input and Output	107
The .PRINTER Driver	108
The .NULL Driver	111
Pipelines	111
The Command Table	112
Command And Utility Reference	114
ALIAS	115
ASM65816	116
ASML	116
ASMLG	119
ASSEMBLE	119
BREAK	119
CAT	120
CATALOG	120
CHANGE	123
CMPL	123
CMPLG	123
COMMANDS	124
COMPACT	124
COMPILE	125
COMPRESS	125
CONTINUE	125
COPY	126
CREATE	128
CRUNCH	128
DELETE	128
DEREZ	129
DEVICES	131
DISABLE	131
DISKCHECK	132
DUMPOBJ	133
ECHO	137

Table of Contents

EDIT	138
ELSE	138
ENABLE	138
ENTAB	139
END	139
ERASE	139
EXEC	140
EXECUTE	140
EXISTS	140
EXIT	141
EXPORT	141
EXPRESS	141
FILETYPE	142
FOR	143
HELP	144
HISTORY	144
HOME	145
IF	145
INIT	145
INPUT	147
LINK	147
LINKER	148
LOOP	149
MACGEN	149
MAKEBIN	150
MAKELIB	151
MOVE	152
NEWER	153
PREFIX	154
PRODOS	154
QUIT	154
RENAME	155
RESEQUAL	155
REZ	156
RUN	156
SET	156
SHOW	157
SHUTDOWN	157
SWITCH	158
TEXT	159
TOUCH	159
TYPE	159
UNALIAS	160
UNSET	160
*	160
Chapter 13 – The Text Editor	161
Modes	161
Insert	161

Table of Contents

Escape	162
Auto Indent	162
Select Text	162
Hidden Characters	163
Macros	164
Using Editor Dialogs	165
Using the Mouse	167
Command Descriptions	167
About	167
Beep the Speaker	167
Beginning of Line	168
Bottom of Screen / Page Down	168
Copy	168
Close	168
Cursor Down	168
Cursor Left	168
Cursor Right	168
Cursor Up	168
Cut	169
Define Macros	169
Delete	169
Delete Character	169
Delete Character Left	169
Delete Line	169
Delete to EOL	169
Delete Word	170
End of Line	170
Help	170
Insert Line	170
Insert Space	170
New	170
Open	171
Paste	172
Quit	172
Remove Blanks	172
Repeat Count	172
Return	172
Save As	173
Save	173
Screen Moves	173
Scroll Down One Line	174
Scroll Down One Page	174
Scroll Up One Line	174
Scroll Up One Page	174
Search Down	174
Search Up	175
Search and Replace Down	175
Search and Replace Up	176
Select File	176

Table of Contents

Set and Clear Tabs	177
Shift Left	177
Shift Right	177
Switch Files	177
Tab	178
Tab Left	178
Toggle Auto Indent Mode	178
Toggle Escape Mode	178
Toggle Insert Mode	178
Toggle Select Mode	178
Top of Screen / Page Up	179
Undo Delete	179
Word Left	179
Word Right	179
Setting Editor Defaults	179
 Chapter 14 – The Link Editor	 183
Overview	183
The Link Edit Process	184
Object Modules Created by the Assembler	184
Subroutine Selection	185
Link Edit Command Parameters	186
Specifying the Keep Name with a Shell Variable	187
Specifying the File Type with a Shell Variable	187
Link Editor Output	188
Output With -S and -L Options	189
The Segment Table	189
Global Symbol Table	189
Program Segmentation	189
Creating Library Files	191
Linker Script Files	192
 Chapter 15 – The Resource Compiler	 195
Overview	195
Resource Decompiler	195
Type Declaration Files	195
Using the Resource Compiler and DeRez	196
Structure of a Resource Description File	196
Sample Resource Description File	197
Resource Description Statements	198
Syntax Notation	198
Include – Include Resources from Another File	198
Read – Read Data as a Resource	200
Data – Specify Raw Data	200
Type – Declare Resource Type	201
Symbol Definitions	209
Delete – Delete a Resource	209
Change – Change a Resource's Vital Information	209
Resource – Specify Resource Data	210

Table of Contents

Labels	212
Built-in Functions to Access Resource Data	213
Declaring Labels Within Arrays	213
Label Limitations	214
An Example Using Labels	215
Preprocessor Directives	215
Variable Definitions	216
If-Then-Else Processing	216
Printf Directive	217
Include Directive	219
Append Directive	220
Resource Description Syntax	220
Numbers and Literals	221
Expressions	221
Variables and Functions	222
Strings	224
Escape Characters	225
Using the Resource Compiler	226
Resource Forks and Data Forks	226
Rez Options	227
 Chapter 16 – GSBug Debugger	 229
Part 1: Getting Started	230
Debugger Restrictions	230
Installing the Init Version on Your Startup Disk	231
Entering the Init Version	232
Entering the Application Version by Launch	233
Setting the Prefix	235
Displaying the Version Number	235
Configuring the Debugger	235
Debugger Display Screens	236
Switching Displays	237
Selecting Displays	237
The Master Display	239
Test-Program Display	240
Memory Display	240
Displaying Memory on the Command Line	242
Direct-Page Display	243
Help Screens	243
Stepping Through Your Program	243
Printing Debugger Screens	243
Using Monitor Routines	244
Quitting the Debugger	244
Part 2: Using GSBug	245
Getting Help	245
Running Your Program	246
Single-Step and Trace Modes	246
Saving and Viewing a Trace History File	249
Real-Time Mode	250

Table of Contents

The Command Filter	251
Memory Protection	251
Breakpoints	252
Debugging Segmented Programs	253
Watching a Running Disassembly	254
Using Breakpoints	255
Using Memory-Protection Ranges	256
Debugging Multi-Language Programs	257
Part 3: GSBUG Subdisplay and Command Reference	257
Register Subdisplay	257
Debugger Registers	258
65816 Registers	259
Emulation Mode Flag	263
Disassembly Mode Flag	263
Altering the Contents of Registers	263
Stack Subdisplay	265
Disassembly Subdisplay	266
Entering Instructions into the Mini-Assembler Display	268
Determining Instruction Length	269
Displaying Toolbox Instructions and GS/OS Calls	270
Displaying Tool Call Information	270
Displaying Instructions During Trace and Single-Step Modes	271
RAM Subdisplay	272
Breakpoint Subdisplay	274
Altering the Contents of the Breakpoint Subdisplay	275
Memory Protection Subdisplay	276
Command Line Subdisplay	279
Entering Commands	279
Viewing Memory with Templates	280
Altering the Contents of Memory	282
Hexadecimal-Decimal Conversions	284
Evaluation of Expressions	285
Configuring the Master Display	286
Saving a Display Configuration	287
Command Line Commands	288
Part 4: Loader Dumper	291
Dump Memory Segment Table	291
Dump Path Name Table	292
Dump Jump Table	293
Dump Loader Globals	295
Dump GS/OS Packets	296
Dump File Buffer Variables	296
Get Load Segment Information	297
Get User ID Information	297
Chapter 17 – Running the Assembler	299
Introduction	299
Shell Commands That Assemble A Program	299
Assembler Command Options	300

Table of Contents

Assembler Directives Global In Scope	303
The Assembly Process	304
Pass One	304
Pass Two	304
Stopping the Listing	304
Terminal Errors	305
The Assembly Listing	305
Screen Listings	305
Printer Listings	306
Chapter 18 – Coding Instructions	307
Types of Source Statements	307
Comment Lines	307
The Blank Line	307
The Characters *	307
The Period	308
Instructions	308
The Label	308
The Operation Code	308
The Operand Field	309
Instruction Operand Format	309
Expressions	311
The Comment Field	315
Chapter 19 – Assembler Directives	317
Introduction To Assembler Directives	317
Descriptions of Directives	317
ABSADDR	318
ALIGN	318
ANOP	319
APPEND	319
CASE	319
CODECHK	319
COPY	319
DATA	320
DATACHK	320
DC	320
Integer	321
Address	322
Reference an Address	322
Soft Reference	322
Hexadecimal Constant	322
Binary Constant	323
Character String	323
Floating Point	323
Double Precision Floating Point	324
Extended Precision Floating Point	324
DIRECT	325
DS	325

Table of Contents

DYNCHK	325
EJECT	326
END	326
ENTRY	326
EQU	326
ERR	327
EXPAND	327
GEQU	327
IEEE	328
INSTIME	328
KEEP	328
KIND	328
LIST	329
LONGA	329
LONGI	329
MEM	329
MERR	330
MSB	330
NUMSEX	330
OBJ	330
OBJCASE	330
OBJEND	331
ORG	331
PRINTER	332
PRIVATE	332
PRIVDATA	332
RENAME	332
SETCOM	333
START	333
SYMBOL	333
TITLE	334
USING	334
65C02	334
65816	334
Chapter 20 – Macro Language and Conditional Assembly Directives	335
The Macro File	335
Writing Macro Definitions	335
Symbolic Parameters	337
Positional Parameters	337
Keyword Parameters	339
Subscripting Parameters in Macro Call Statements	339
Explicitly Defined Symbolic Parameters	340
Predefined Symbolic Parameters	341
Sequence Symbols	343
Attributes	343
C: Count	343
L: Length	344
S: Setting	344

Table of Contents

T: Type	345
Conditional Assembly and Macro Directives	346
ACTR	346
AGO	347
AIF	348
AINPUT	348
AMID	349
ASEARCH	349
GBLA	350
GBLB	350
GBLC	350
GEN	350
LCLA	351
LCLB	351
LCLC	351
MACRO	351
MCOPY	351
MDROP	352
MEND	352
MEXIT	352
MLOAD	352
MNOTE	353
SETA	353
SETB	353
SETC	354
TRACE	354
Chapter 21 – Introduction to the Macro Libraries	355
GS/OS Macros	356
Macro Naming Conventions	356
Inside the GS/OS Macros	357
Tool Set Macros	357
Addressing Modes	359
Data Types	359
Two-Byte Integers	360
Four-Byte Integers	360
Eight-Byte Integers	360
Character	360
Strings	361
Boolean Variables	361
Memory Usage	361
Side Effects	361
Chapter 22 – Mathematics Macros	363
ADDx	364
CMPx	365
DIVx	366
MODx	367
MULx	368

Table of Contents

RANx	369
SIGNx	370
SQRTx	371
SUBx	372
Chapter 23 – Input and Output Macros	373
ALTCH	373
BELL	374
CLEOL	374
CLEOS	375
COUT	376
GETx	377
GOTOXY	379
HOME	380
NORMCH	380
PRBL	381
PUTx	382
PUTCR	384
Chapter 24 – Shell Calls	385
Making a Shell Call	385
The Control Record	385
Types of Parameters	385
Register Values	386
ChangeVector	388
ConsoleOut	390
Direction	391
Error	393
Execute	394
ExpandDevices	396
Export	398
FastFile	400
GetCommand	405
GetIODevices	407
GetLang	408
GetLInfo	409
InitWildcard	414
NextWildcard	417
PopVariables	419
PushVariables	420
ReadIndexed	421
ReadVariable	423
Redirect	425
Set	427
SetIODevices	429
SetLang	431
SetLInfo	432
SetStopFlag	437
StopGS	438

Table of Contents

UnsetVariable	439
Version	440
Chapter 25 – Miscellaneous Macros	441
ASL2	441
BGT	442
BLE	442
BUTTON	443
CLSUB	443
CNV _{xy}	444
CSUB	445
DB _{cn}	445
DEC2	446
DEC4	446
DOSIN	447
DOSOUT	448
DSTR	449
DW	450
INC2	451
INC4	451
J _{cn}	452
LA	453
LLA	454
LM	455
LONG	456
LRET	456
LSR2	457
LSUB	457
MASL	458
MLSR	458
MOVE	459
MOVE4	460
PH _x	461
PL _x	462
PREAD	463
RESTORE	464
RET	464
SAVE	465
SEED	466
SHORT	467
SOFTCALL	468
SUB	469
Appendix A – Error Messages	471
Error Levels	471
Recoverable Assembler Errors	471
ACTR Count Exceeded	472
Address Length not Valid	472
Addressing Error	472

Table of Contents

Duplicate Ref in MACRO Operand	472
Duplicate Segment	473
Expression Too Complex	473
Invalid Operand	473
Label Syntax	474
Length Exceeded	474
Macro File Not In Use	474
MACRO Operand Syntax Error	474
Missing Label	474
Missing Operand	474
Missing Operation	475
Misplaced Statement	475
Nest Level Exceeded	475
No MEND	475
Numeric Error in Operand	476
Operand Syntax	476
Operand Value Not Allowed	476
Sequence Symbol Not Found	477
Set Symbol Type Mismatch	477
Subscript Exceeded	477
Too Many MACRO Libs	477
Too Many Positional Parameters	477
Undefined Directive in Attribute	478
Unidentified Operation	478
Undefined Symbolic Parameter	478
Unresolved Label not Allowed	478
Terminal Assembler Errors	478
AINPUT Table Damaged	479
File Could not be Opened	479
GS/OS Errors	479
Keep File Could Not be Opened	479
Out Of Memory	479
No END	479
Unable to Write to Object Module	479
Recoverable Linker Errors	479
Addressing error	480
Address is not in current bank	480
Address is not in zero page	480
Alignment and ORG conflict	480
Alignment factor must not exceed segment align factor	480
Code exceeds code bank size	481
Data area not found	481
Duplicate label	481
Duplicate segment	481
Expression operand is not in same segment	481
Illegal {AuxType} shell variable [4]	481
Illegal {KeepType} shell variable [4]	481
Only JSL can reference dynamic segment	482
ORG Location has been passed	482

Table of Contents

Relative address out of range	482
Segment types conflict	482
TempORG not supported	482
Unresolved reference	483
Terminal Linker Errors	483
Could not find library header in filename.	483
Could not overwrite existing file: filename.	483
Expression too complex in filename.	483
File read error	483
File not Found filename.	483
File write error.	484
Invalid dictionary in filename.	484
Must be an object file: filename.	484
Only one script file is allowed.	484
Script error: link aborted.	484
Stopped by open-apple .	484
Appendix B – File Formats	487
Overview	487
Text Files	487
Object Modules	487
Executable Files	497
Appendix C – Custom Installations	499
Installer Scripts	499
New System	499
ORCA Icons	500
.PRINTER and .NULL	500
GSBug	500
ORCA Pascal, C, Asm Libraries	500
Update System, No Editor	501
RAM Disks	501
Details About Configuration	501
Appendix D – Licensing	505
Using the .PRINTER Driver	505
Macros	505
Using SYSLIB	505
Appendix E – Differences Between ORCA/M 2.0 and ORCA/M 1.0	507
GS/OS	507
Path Names	507
Standard I/O	508
Numbered Prefixes	508
Shell	508
Command Line Length	508
Shell Variable Length	509
New .PRINTER Driver	509
New .NULL Driver	509

Table of Contents

Shell Prefix	509
Larger Default Stack	509
New Shell Variables	509
New Command Line Editor	509
New or Changed Commands	510
Editor	511
ASM65816	511
LINKER	511
Rez Compiler	511
GSBug	512
New Utilities	512
Macros	512
Index	513

Welcome To ORCA

ORCA is an assembly language development system designed for the Apple IIGS computer. It is a combined command processor, screen editor, sophisticated macro assembler, linker, library and support programs. ORCA will allow you to create and execute your assembly language programs with ease. This preface will guide you through the preliminaries of using ORCA.

- The express installation and set up of ORCA.
- How to use this manual.
- The hardware and software needed to run ORCA.
- Additional reading and reference.

Express ORCA Start Up

The minimum hardware required to run ORCA/M 2.0 under System Disk 5.0.4 is 1.25M with ROM 01, and 1.125M with ROM 03 Apple IIGS computers. You must have at least one 3.5 floppy disk drive. As ORCA and Apple's operating system evolve, these requirements may change; if so, you can find details in a file called Release.Notes on the Extras disk.

The fastest way to get up and going is to execute the ORCA.Sys16 file from the ORCA/M program disk. Before starting, though, use a copy program to make a working version of ORCA. *Never use your original ORCA disk to run ORCA!* You are now ready for Chapter 1. Be sure to look through this preface at some later time.

If you will be using ORCA/M from a hard disk, you can install it quickly by first creating a folder called ORCA, then running Apple's Installer from the Extras disk. This installer gives you a number of installation options; you should set the folder to the newly created ORCA folder, then select the New System installation script. When the installation is complete, your ORCA folder will contain the file ORCA.Sys16; run this file to start ORCA/M. If you have any questions about the installation scripts, check Appendix C for more detailed information.

About This Manual

This manual will guide you through all aspects of the ORCA environment. To make it easy for you to learn ORCA, this manual has been divided into four major sections. The first part is called the "User's Guide." It has instructions for starting ORCA as well as an introduction to creating assembly language programs under ORCA. The second part is entitled "Operating Environment Reference Manual." It is a working reference to provide you with in-depth knowledge of the command processor, the editor, the program linker, the program libraries, the resource compiler, the GSBUG debugger, and program execution. Part three is the "Assembler

User's Manual

Reference Manual." It contains information about the assembler directives. The last part is called the "Macro Reference Manual." It lists and describes all the macros provided with ORCA.

This manual is not designed to teach you assembly language programming. Basic concepts about this type of programming are necessary to create useful, efficient programs. If you are new to assembly language programming, pick up a beginner's book on this subject on your next visit to your computer store. See the reading list at the end of the preface for some suggestions.

If you are new to ORCA, start at the beginning and carefully read the "User's Manual." It was written with you in mind. Work all the examples, and be sure that you understand the material in each chapter before leaving it. ORCA is a big system, and like any sophisticated tool, it takes time to master. The large size of this manual is a reflection of the power ORCA can offer you.

Visual Cues

In order to delineate between information that this manual provides and characters that you type or characters that appear on your computer screen, special type faces are used. When you are to enter characters, the type face **looks like this**. When you are supposed to notice characters displayed on the computer screen they look like this.

What You Need

In order to use ORCA, you must have the following hardware and software. A list of Apple IIGS manuals that you will find useful is given at the end of the preface.

- An Apple IIGS computer.
- A minimum of 1.25M of RAM if your Apple IIGS uses ROM 01, or a minimum of 1.125M if your Apple IIGS uses ROM 03.
- A 3.5 inch disk containing the ORCA system.
- At least one 3.5 inch disk drive. However, it is strongly recommended that you have two 3.5 inch disk drives, or one 3.5" disk and a hard disk.

The following hardware is highly recommended, especially if you intend to do multiple-language development or to develop large programs:

- A hard disk.
- A printer.
- 2M or more of RAM.

Additional Reading And Reference

Books about about the 65816 (the microprocessor used in the Apple IIGS) are available from several publishers; a few are listed below. There are several manuals published by Apple Computers which provide you with additional knowledge of the Apple IIGS computer. Depending

Preface

upon your applications you may need to refer to some of these manuals. For your convenience, a reading list is supplied listing the available Apple IIGS reference manuals.

Reading List

The following books introduce assembly language programming and how it relates to the 65816. Some are good for beginners, others are best for advanced programmers. You should visit your bookstore to pick out the one best suited for your needs. The source code used in these books was generated from ORCA.

Apple IIGS Assembly Language Programming

Leo J. Scanlon

Bantam Books, Toronto, New York, London, Sydney, Auckland

Programming the 65816

William Labiak

SYBEX, Berkley, California

Programming the 65816

Including the 6502, 65c02, and 65802

David Eyes and Ron Lichty

Brady Prentice Hall Press, New York, New York

Programming the Apple IIGS in Assembly Language

Ron Lichty and David Eyes

Brady Prentice Hall Press, New York, New York

65816/65802 Assembly Language Programming

Michael Fischer

Osborne McGraw-Hill, Berkley, California

The following books are in the Apple IIGS Technical Manual Suite. These books are written by the staff at Apple Computer, Inc., and published by Addison-Wesley.

Technical Introduction to the Apple IIGS

A good basic reference source for the Apple IIGS.

Apple IIGS Hardware Reference

Apple IIGS Firmware Reference

These manuals provide information on how the Apple IIGS works.

Programmer's Introduction to the Apple IIGS

Provides programming concepts for the Apple IIGS.

Apple IIGS Toolbox Reference Volume I

Apple IIGS Toolbox Reference Volume II

Apple IIGS Toolbox Reference Volume III

These volumes provide essential information on how tools work.



User's Manual

Apple II GS ProDOS 8 Reference

Apple II GS ProDOS 16 Reference

GS/OS™ Reference Manual

These manuals provide essential information on file operations.

Chapter 1

Getting Started

This chapter gives you a brief overview of ORCA by stepping through the development of a simple program. Only the most commonly used commands and features are described in this chapter. Do not be concerned about features mentioned that you are unfamiliar with. Future chapters will cover all the topics covered in this chapter in much more detail. You will be introduced to the three main programs contained in ORCA. The topics covered are:

- How to enter and execute ORCA shell commands.
- How to use the text editor.
- How to assemble and link an example program.

The ORCA Shell

The shell, or command processor, is the interface between you and ORCA. It is through this shell that you command ORCA to start an editing session, manipulate files and directories, or to compile and link a program. When ORCA starts up, it is the shell program that becomes active, waiting for and then responding to your commands. In this chapter only a brief discussion of the shell is given. You will learn more about the ORCA shell in Chapter 2.

Shell Commands

As stated above, the shell is a resident program when ORCA is active. You will know that the shell is running by the pound sign (#) that is used for the shell prompt. At this point go ahead and start up ORCA if you have not already done so. You should see the prompt on the left-hand edge of the screen. When you see #, ORCA is waiting for a command. Shell commands are contained in a command table internal to ORCA. When you enter a command, ORCA checks this table against your command. If ORCA can't find the command, it checks the current directory for a file by the name typed. If one is found, it is executed. If a file is not found, the shell will respond with the message "File not found: <file name>". The file name listed will be the full path name of the program the shell tried to execute. (Note: If you are not familiar with files and directories, please see the first three pages of Chapter 5.)

In preparation for the sample program, you will need to enter a couple of shell commands. The program is in assembly language, so you have to tell ORCA to expect an assembly language program. This is accomplished by the command

```
asm65816
```

User's Manual

Another thing you should tell ORCA is the directory you will want the sample program contained in. As an example, suppose the name of the volume you wanted was myprogs. Then the command you would give ORCA would be

```
prefix :myprogs
```

This command causes the current directory to be changed to :myprogs.

Your next step is to use the editor and enter the sample program.

The Editor

The ORCA editor is a full-screen text editor, with considerable text-manipulation facilities. With the editor you can create new files or edit old ones.

Calling The Editor

At this point you should still be in the shell. To invoke the editor you need to give ORCA the EDIT command with the name of the file you want the program source to be in. For example, if you want to have the sample program in a file named hello.asm you would enter

```
edit Hello.asm
```

While you can use any legal file name for your program, following a few simple naming conventions will pay off in the long run. Until you know enough about how the system works to see all of the various files it creates, we suggest choosing a name for your program that is relatively short, and in no case longer than nine characters. The name of the source file (the program you type in using the editor) should be the name of the program, followed by the letters '.asm'. In this example, the name of the program will be hello, so the name of the program we create with the editor is hello.asm.

After typing this command, the editor will become active. You should see a clear screen with a banner at the bottom. This banner will give you useful information about the cursor location, what file you are editing and what type of editing you are doing. The cursor will be located in the upper corner of the screen in column one, line one. In order to move the cursor around, there are some keys that will provide movement of the cursor. Any character key will display its respective character when pressed and then move the cursor into the next column to the right. The delete key will move the cursor back to the left, erasing characters as it goes. The tab key will move the cursor in a predetermined manner. The first tab is set for column ten. The second tab is located at column sixteen. These tab positions make it easy to enter assembly language programs. The arrow keys will move the cursor around as indicated by each key. The return key will move to the beginning of the next line. If you make a mistake in typing, position the cursor at the error and type over it.

Go ahead and start up an editor session and enter the sample program given below.

```

Main      keep    Hello
          mcopy   Hello.Macros
          start
          phk
          plb
          puts    #'Hello world! ',cr=t
          lda     #0
          rtl
          end

```

The format of the program is important. Make sure there is at least one space between op codes and operands - e. g. between KEEP and HELLO. Be sure labels, like MAIN, start in the first column, and that operation codes, like PHK, don't. There must be at least one space between MAIN and START. Do not put any spaces, except within quotes, in the operand - e.g. don't put a space between 'Hello, World!' , and cr=t. The easiest way to get a clean listing is by using the predefined tab stops.

Press **⌘S** to save your file, then **⌘Q** to quit the editor.

The discussion of the editor was very simplified. It was only an example to get you started. You will learn more about the editor in Chapter 3.

For the most part, we'll leave a discussion of the program itself for Chapter 4, but you can see more in the way of the naming convention mentioned earlier in this program. The first line tells the assembler to keep the program using the file name Hello; this will be the name of the finished, executable program. The second line tells the assembler that it will use a macro file called Hello.Macros. As you can see, this means there will be at least three files associated with your program, but using a naming convention lets you quickly see that all three files belong to the program Hello, and clearly marks one file as the assembly language source file (the one ending with .asm) and another as the macro file (the one ending in .macros). The macro file doesn't exist yet; we'll create it in the next section.

Assembling and Linking a Program

ORCA uses a single format for object files and a single set of commands for compiling or assembling programs written in any source language. Therefore, you can write different modules or routines of your program in different languages, and compile, link, and run the program all in one step. For the vast majority of programs written in the ORCA environment, the compiler and linker defaults are quite adequate.

Since this is an ORCA assembly language program that uses macros, you have to create a macro file for use by the assembler. To create a macro file for your program, you need to know three things: the name of your program's source file, in this case hello.asm; the name of the macro file you want to create, in this case hello.macros; and the location of the macro library files you would like to scan. Where the macros are depends on whether you are running on a hard disk or from floppies.

From a Hard Disk

The macro library files that come with ORCA/M are moved to the ORCA libraries folder when you install ORCA/M on your hard disk. There are actually two macro folders. The first, AInclude, contains the macros and equate files that Apple Computer ships with Apple Programmer's Workshop (APW). The other folder, called ORCAInclude, contains the various

User's Manual

macros described in this manual, plus macros for Apple's toolbox that allow you to use parameters.

Our program uses some of the output macros from the ORCA/M macro library, located in the folder ORCAInclude. To create the macro file Hello.Macros, enter the following command:

```
macgen hello.asm Hello.Macros 13:orcainclude:m=
```

This command uses a utility called MacGen. This utility reads your assembly language program, which is the first file name you enter, looking for any instructions that are not 65816 assembly language instructions or predefined assembler directives. It creates a macro file called Hello.Macros, reading the macros from the files in the ORCAInclude macro library. All macros supplied with ORCA/M, both those we wrote and the ones Apple wrote, are in files with a file name starting with the letter M. The command you just typed in tells MacGen to look at all files in the folder that have a first character of M in the file name, so MacGen will scan all of the libraries until it finds the macros it needs.

From Floppy Disks

The macro library files that come with ORCA/M are located on the Extras disk. It would be better to put them in the libraries folder of the ORCA disk, but there isn't room for everything we would like to put on the ORCA disk.

You have several options for dealing with the macros, but they boil down to three real possibilities: either delete some files from the copy of the ORCA disk you are using to make room for the macro libraries on your ORCA disk; or copy the macros to the same disk you are using to save your program's source files; or if you have two 3.5 inch disk drives plus another drive for your programs, put both the ORCA disk and the Extras disk in the drives at the same time. If you decide you want to copy the macro files to the disk where your program's source code is located, keep in mind that you can copy only those files you need. For this example, you only need the file :ORCA.Extras:Libraries:ORCAInclude:M16.ORCA. (The full path name will help you find the specific file we're using, but you can put the M16.ORCA file anywhere you like.)

The macros supplied with ORCA/M are on the Extras disk, located in the folders :ORCA.Extras:Libraries:AInclude and :ORCA.Extras:Libraries:ORCAInclude. The folder AInclude contains the macros and equate files that Apple Computer ships with Apple Programmer's Workshop (APW). The ORCAInclude folder contains the various macros described in this manual, plus macros for Apple's toolbox that allow you to use parameters.

For the rest of this section, we'll assume that the M16.ORCA file has been moved to the same disk where you have saved hello.asm. If it is on another disk, substitute the full path name for the m= parameter in this example.

To create the macro file Hello.Macros, enter the following command:

```
macgen hello.asm Hello.Macros m=
```

This command uses a utility called MacGen. This utility reads your assembly language program, which is the first file name you enter, looking for any instructions that are not 65816 assembly language instructions or predefined assembler directives. It creates a macro file called Hello.Macros. All macros supplied with ORCA/M, both those we wrote and the ones Apple wrote, are in files with a file name starting with the letter M. The command you just typed in tells MacGen to look at all files that have a first character of M in the file name, so MacGen will scan all of the macro files in the current folder until it finds the macros it needs.

Assembling the Program

To assemble, link, and execute the program, enter:

```
run hello.asm
```

Following the diagnostic output of the the assembler and linker, the words `Hello world!` should be written to the screen. If not, check your source file for errors and try again.

While it doesn't take long to assemble or link the program, you may have noticed a delay as the assembler and linker were loaded from disk. Once loaded, the assembler and linker stay in memory unless the memory is needed for some other purpose. The next time you assemble and link a program, or use MACGEN or the editor, for that matter, the program will not need to be loaded from disk, so things will go much faster.

You now have a file on your work disk called `Hello`. To execute this program again, without reassembling the program, enter `hello` from the ORCA shell command line.

Chapter 2

The Shell

This chapter is an introduction to using the shell. In the first chapter, you learned that the shell was the interface between you and ORCA. Now you will learn how to use a few commands and shell features. Go ahead and start up ORCA if you have not already. Topics you will learn about in this chapter are:

- The shell commands.
- The shell line editor.
- Listing a directory.
- Displaying the contents of a file.
- Changing a file's language.
- Using wildcards in file names.
- Required and optional parameters.
- Types of commands.
- How to set up a printer.
- Printing a file.

The Shell Commands

In Chapter 1, it was mentioned that when you enter commands while in the shell, ORCA checks the command against the command table. If you entered a valid command, the command would be executed. Table 2.1 gives a listing of the commands in the command table, as listed by the HELP command. Go ahead and examine the table - some of the entries will probably look familiar to you.

Available Commands:

ALIAS	ASM65816	ASML	ASMLG	ASSEMBLE	BREAK
CAT	CATALOG	CHANGE	CMPL	CMPLG	COMMANDS
COMPACT	COMPILE	COMPRESS	CONTINUE	COPY	CREATE
CRUNCH	DELETE	DEREZ	DEVICES	DISABLE	DISKCHECK
DUMPOBJ	ECHO	EDIT	ELSE	ENABLE	ENTAB
END	ERASE	EXEC	EXECUTE	EXISTS	EXIT
EXPORT	EXPRESS	FILETYPE	FOR	GSBUG	HELP
HISTORY	HOME	IF	INIT	INPUT	LINK
LINKER	LOOP	MACGEN	MAKEBIN	MAKELIB	MOVE
NEWER	PREFIX	PRODOS	QUIT	RENAME	RESEQUAL
REZ	RUN	SET	SHOW	SHUTDOWN	SWITCH
TEXT	TOUCH	TYPE	UNALIAS	UNSET	*

Table 2.1 The ORCA Shell Commands

Listing a Directory

In order to inspect a directory for the files or subdirectories it contains, you would issue the CATALOG command. For example, if you wanted to inspect the :ORCA directory, enter the command

```
catalog :orca
```

Go ahead and try the CATALOG command now. The directory listing for your ORCA program disk will look similar to the directory listing displayed in Table 2.2. Don't be alarmed if the directory you display is not identical to the one in the table. The heading of the directory listing gives attributes for each directory entry.

```
:ORCA:=
```

Name	Type	Blocks	Modified	Created	Access	Subtype
ORCA.Sys16	S16	156	15 Oct 91	15 Oct 91	DNBWR	\$DB01
Shell	DIR	1	15 Oct 91	15 Oct 91	DNBWR	\$0000
Languages	DIR	1	15 Oct 91	15 Oct 91	DNBWR	\$0000
Libraries	DIR	1	15 Oct 91	15 Oct 91	DNBWR	\$0000
Utilities	DIR	1	15 Oct 91	15 Oct 91	DNBWR	\$0000
Icons	DIR	1	15 Oct 91	15 Oct 91	DNBWR	\$0000
Samples	DIR	1	15 Oct 91	15 Oct 91	DNBWR	\$0000
Blocks Free:	37	Blocks used:	1563	Total Blocks:	1600	

Figure 2.2 Sample Listing Produced by the CATALOG Command

The CATALOG command gives a lot of information about the listed directory. Don't be concerned if you don't know what every attribute in this listing means; further reading will explain them. Chapter 12 has the information summarized if you are interested in reading it now.

To display the contents of a file on the screen, you would issue the `TYPE` command. For example, to see what's in the `Release.Notes` file on the `:ORCA.Extras` disk, you would first place the extras disk on-line, and then enter the command

```
type :orca.extras:release.notes
```

To pause the display, press any key. To stop the display, press `⌘.` (hold down the open-Apple key and then press the period key).

Another useful command is `CHANGE`. You would use this command to change the language associated with a file. ORCA assigns a language to every source and text file. You can see a file's language by doing a `CATALOG` of the file's directory, and examining the `SUBTYPE` field of the catalog heading. For example, suppose you had created a new file named `File.asm` with the editor, and after doing a `CATALOG` of the directory you discovered that the file's subtype was `TEXT`. ORCA uses the subtype field to denote a file's language. You wouldn't be able to assemble `File.asm` because the assembler insists that its language be `ASM65816`, and not `TEXT`. In order to change `File.asm`'s language, you would use the `CHANGE` command:

```
change file.asm asm65816
```

Note that the `CHANGE` command expects a file name as its first parameter, and a language name as its second. `ASM65816` is the name of ORCA's assembler.

The Shell Line Editor

When you enter commands in the shell, you can make your typing chores somewhat easier by using the features of the line editor. The features covered here serve only as an introduction to the line editor. It will be covered in greater detail in Chapter 5.

Entering Commands

So far, you have issued commands by typing the entire command name. It is not necessary for you to type in the entire command, however. Type in the first letter or first few letters of the command, then press the `RIGHT-ARROW` key (`->`). The shell consults the command table, and prints out the full command name of the first command it finds that matches the letters you typed. For example, type the following (not pressing the `RETURN` key yet):

```
H ->
```

For this example, you will issue the `HELP` command. This is a very useful command in that it gives you instant information about ORCA commands. The shell finds the first command name in the command table that begins with `H` and prints the full command name:

```
HELP
```

When you press `RETURN`, the entire command line is sent to the command interpreter, regardless of the location of the cursor on the command line. Go ahead and try it. You will get a listing of all the shell commands available to you, as seen in table 2.1. From here you can use the help command again for each of the command table entries. For example, if you wanted information on the `SHOW` command, enter

User's Manual

HELP SHOW

There is one command for which no help is available. This is the *, or comment, command. The reason is simple: while ORCA can use any character as a command name, GS/OS will not allow a file named * when using the ProDOS FST. Since the help file for a command has a file name that matches the command name, a help file for the command * could not be created.

Spend some time investigating these commands by using the help feature. Don't forget to use the right arrow!

Scrolling Through Commands

You can press the UP-ARROW and DOWN-ARROW keys to scroll through the last 20 commands that you have entered. You can then modify a previous command and press RETURN to reenter it. Each time you enter or reenter a command, that command is appended to the 20-command list, with the first command in the list being lost.

To try this out, enter the following sequence of commands. Be sure your :ORCA disk is on-line.

```
#prefix :orca
#catalog

[directory listing printed]

#[press UP-ARROW key]

#catalog utilities

[directory listing printed]

#[press UP-ARROW key]

#catalog utilities:help

[directory listing printed]

#[press UP-ARROW key]

#catalog utilities:help [press UP-ARROW key]

#catalog utilities [press UP-ARROW key]

#catalog
```

The 20-command list is circular; that is, once you have used the UP-ARROW 20 times to scroll through the 20 previous commands, pressing the UP-ARROW one more time prints the last command entered again. Experiment with the command-line scrolling; you will find that this function can save you a lot of time in entering long or complex commands.

Using Wildcards

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. The privilege flags can be examined in the catalog command display. The flags are represented by characters under the access attribute. See Table 2.1 for an example or enter `CATALOG`. The `ENABLE` and `DISABLE` commands are 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 the access column of each directory entry. This means that you can no longer delete the files. Now, enable the delete privilege for the `ORCA.Sys16` file. Since the `ORCA.Sys16` file is the only one that starts with the character `O`, you can do this by typing

```
ENABLE D O*
```

The wildcard matches all of the characters after `O`.

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 `ORCA.Sys16`, you can specify the file as `=sys16`. 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 you 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`, `COPY`, and `MOVE` 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, you 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 shell, so it is always available. This is the type of command supported by BASIC under both DOS and ProDOS.

A second kind of command is the language. ORCA is set up to support multiple languages.

The last kind of command is the utility. Utility commands are commands that use a separate program to do their function. INIT, which is used to format a disk, is an example. When you use a utility command, ORCA must find and load a completely separate program. The reason that utility commands are used at all is to be able to add commands to the system that may need to change more rapidly than the shell. In addition, you will be able to add your own commands to the system. This is discussed more fully in Chapter 9.

Chapter 3

The Editor

Chapter 1 provided you with only a taste of the features provided by the ORCA editor. In this chapter we will cover the editor features in much greater detail. As new editor features are covered, there are topical examples for you to try. Topics covered in this chapter are:

- Editor modes.
- Entering text.
- Moving through a file.
- Modifying text.
- Search / search and replacement.
- Cut, copy, paste.

The editor has over fifty commands. Many of the features will not be covered now. When you complete this chapter, you may want to look through Chapter 13, which covers all of the editor features.

Starting The Editor

In order to work through all of the commands covered in this chapter, a file with a fair amount of text will be needed so that you can see how the commands work. For this purpose, we will use an existing file which contains some type declarations for the resource compiler. This file will be important when you start writing programs, so we will need to exercise some care. When it comes time to exit the editor, don't save the changes! The last section of this chapter will guide you through closing the file. If you are nervous, remember that you are working from a copy of the original disk, so you can always go back to the original if you mess up. You can also write protect your disk while you work through this chapter, but remember to set the write protect tab back so the disk can be modified before moving on to future chapters.

Let's get started. Go ahead and start up the editor with the command

```
edit 13:rinclude:types.rez
```

Help

At any time, you can press `Ctrl?`, which causes the editor to display a help file. This file has a brief list of the commands available in the editor. You can use the cursor movement commands, like the arrow keys, to scroll through the help file. RETURN or ESC returns you to the file you are editing.

Editor Modes

With the editor running, you will see an eighty-column screen with text displayed on twenty-two lines. At the bottom of the screen is a reverse-video banner containing the current editor mode. You should see the message `Mode: EDIT`. The editor has several modes which, when activated, will allow you to use different editing features. There are five other modes in addition to the edit mode. In this chapter we will only cover four modes: edit, escape, insert, and select. The features associated with these modes will be introduced later in this chapter. The two modes not discussed in this chapter are auto indent and hidden characters. These are covered in Chapter 13.

Entering Text

The edit mode is where you will do most of the text entry. Also, most of the commands are active in the edit mode. Having just started the editor, the cursor is positioned at the start of line one. Go ahead and enter a few words. As you enter text, the cursor will over-strike any characters it encounters. Keep typing characters. When you get to the edge of the screen, the editor will shift over, showing several new columns of text. The ruler at the bottom of the page shows that you are now past column 80. You can keep going this way until you get to column 255, when the line will automatically wrap.

Moving Through A File

There are many ways to move through a file. We should classify the range of movement in order to present the commands in a rational order. Let's say that:

- Small movement - a few characters or lines
- Medium movement - moving the range of the display screen (a page)
- Large movements - from the start and end of the file

Let's start small and work up.

Small Movements

For assembly language programming, one of the most frequently used movement commands is the `TAB` key. Press `RETURN` to move the cursor to the start of a line. Now press the `TAB` key. Notice the column position of the cursor. The tab stops are set at columns ten, sixteen, and forty-one. By pressing `CTAB`, you can move back a tab stop. These tab stops are set for entering assembly language programs. You can change the stops to best suit you. See pages 154 and 155.

If you use the `TAB` key in insert mode, which we'll cover in a moment, or if you use the `TAB` key while you are past the last character on the line, it no longer just moves the cursor. Instead, a tab character is inserted in the text file.

The arrow keys are great for moving a few characters to the right and left, or moving between a couple of lines up and down. These keys are really easy to use: the direction of cursor movement is on the key. Hold an arrow key down for automatic repeat. Go ahead and experiment with these

keys. You could use these keys to move anywhere in the file. But you don't have to – there are better ways!

Two commands let you move to the start or end of a line. To move the cursor to the end of a line, use the `↵>`, the end-of-line move command. To move the cursor back to the start of a line, enter the `↵<`, the start-of-line command.

Medium Movement

In this range of movement, you will learn how to move to the top and bottom of a page. As stated earlier, a page is the amount of text visible on your display screen. There are two commands which will let you move the cursor to the top and bottom of the page.

The top of page command, `↵UP-ARROW`, will move the cursor to the first line of the page. Go on and give it a try. Notice that the cursor remained in the column it started in. To move the cursor to the bottom of a page, use `↵DOWN-ARROW`. This command will move the cursor to the last line of the page, the cursor again remaining in the column it started in. There is an extra feature associated with these two commands. Use the `↵DOWN-ARROW` command to position the cursor on the bottom line of a page. Now use the `↵DOWN-ARROW` command a second time. Examine the line count. It has increased by twenty-two lines! What happened is that use of `↵DOWN-ARROW` when the cursor is on the last line of a page will cause the next page to be displayed, and the cursor to be moved to the last line on that page (while preserving the horizontal position of the cursor). Position the cursor on the top line of the current screen and try the `↵UP-ARROW` command. It works in the same way.

Large Movements

As files become larger, you will find the commands which perform large movements convenient. There are two methods to span a large distance in the file.

The first method is called scrolling. Scrolling is by a line or page. Let's start with line scrolling. To use the line scroll command, you need to be in escape mode. By the name, you can see it involves the escape key `ESC`. You enter escape mode by pressing `ESC`. You can exit the mode by pressing `ESC` a second time. This is called toggling the mode. When an escape mode command is used in this manual, we will assume that you will return to the edit mode when you finish the command. Back to the line scroll command. Go ahead and enter the command `ESC↵`. Did you notice what happened? The screen scrolled up one line. "Big deal," you say. "I could use the `RETURN` key for that." Well, here is where escape mode commands become interesting. Integrated with every escape mode command is the repeat count feature. Try the command `ESC10↵`. The cursor moved down ten lines! You can specify any integer in the range 1 to 32767 in any escape mode command. Try it with the scroll up command, `ESC10↵`.

There are also commands that scroll one page without changing the position of the cursor. Try the command to scroll down one page, `↵]`. It has increased the line count by twenty-two lines. Let's try something completely different. This file we have been using is not long (about 550 lines) so let's try to scroll past the end of the file. If we give the command to scroll 300 pages, it would be past the end of the file. Go ahead and enter `ESC 300↵]` and watch what happens. The editor would let us scroll past the end of the file. Now try scrolling past the start of the file. Enter the command `ESC500↵[` and notice what happens. In this case the editor stopped us from going past the beginning of the file, and placed the cursor on the first line of the file.

Another command for large movement allows you to move between eight even intervals in the file. The editor has divided our file into eight approximately equal sections. Using the screen move commands, you can move to the boundaries between these sections. Let's see how these commands work. Go ahead and enter the command `␣1`, and then the command `␣9`. The cursor first jumped to the first character in the file, and then jumped to the last character in the file. These two positions are the boundaries before the first section and after the eighth section. Try the command sequence `␣2` through `␣8` in order to get a feel for these commands. Notice the cursor position after each command. Also, note that these commands are not available in the ESC mode.

Modifying Text

Most of the work that you will do in the editor will be changing existing files. There are several methods available to you. The most often used features are deleting and inserting text. Using these two features, any type of modification to a file could be made.

Deleting Text

There several commands for deleting text; only those most commonly used will be discussed here. The amount of text you need to delete will determine the command you use. Let's go through some of the delete commands available to you, starting with minor deletions of a character or two and working up to many lines.

You may have already used the first delete command, which is the the delete key, or `DELETE`. When you press this key in the edit mode, the cursor will move one column to the left, removing the character which occupied the position to the left of the cursor. Similar to the delete command is the command to delete a character at the cursor. This command is `␣F`. When this command is given, the character covered by the cursor is deleted and the remainder of the line is moved one position to the left. Go ahead and try these commands.

The two commands covered above are fine for deleting a few characters; it would be nice to be able to delete an entire line. This next command deletes text from the cursor to the end of the line. Choose a line in the file which contains some characters. Position the cursor about half way in the line and press the command `␣Y`. The command caused all of the text from the cursor to the end of the line to be deleted. Now, position the cursor on the first position of the line and repeat the command. What happens? The editor deletes all of the characters from the line.

If you want to delete more than one line, use the delete line command. Go ahead and position the cursor at the start of the file. Enter the command `␣T`. Only one line is removed from the file. The command is also available from escape mode, where you may enter a repeat count number and remove as many lines as needed. Try this command out. Be sure and leave some of the file for the next sections!

There is another way to delete text that involves selection the text you want to delete; this will be covered later.

Inserting Text

Our first insert command will allow us to insert spaces. Place the cursor between two characters and enter the command `^SPACE`. A space will now separate the two characters. You could use the automatic repeat count in the escape mode and enter as many spaces as needed.

The second insert command will allow us to insert blank lines. To use this command, place the cursor on the line below the position where the new lines are expected. For example, if you wanted the new lines to appear after line one, you would position the cursor in line two. Now enter the command `^B`. There is your blank line. You can use repeat counts to insert more blank lines.

One of the most 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 lines left over. Typing `^R` removes these lines, beginning at the cursor position and continuing to the first non-blank line. Let's give this a try. Insert fifty blank lines. Now move the cursor down a couple of lines, and type `^R`. The line the cursor is on must be blank, or the command will be ignored.

The last method of inserting is using the insert mode. By using the insert mode, you can position the cursor at any position in the file and insert extra characters. Go ahead and position the cursor between some characters. Toggle insert mode by typing `^E`. Now that you are in the insert mode, you can enter as much new text as needed. Exit insert mode by typing `^E`.

Depending on your background, you may be more comfortable using the insert mode all of the time. You can make the insert mode the default mode; see the description of the SYSTABS file in Chapter 13 to find out how.

Advanced Features

Search / Search With Replace

These features can be a big time saver for you. Using these commands, you can have the editor search any portion of a file for a character string that you specify. You can even have the editor replace the character string with another character string that you specify. For example, if you misspell the word "label" as "lable" throughout your file, you can have the editor replace the misspelled version with the correct version.

Search

This command is used to locate a sequence of characters specified by you. For example, let's suppose that you wished to locate every occurrence of the word `type` in our file. First, position the cursor on the first character of the file. Now, enter the search command `^L`. There are really two search commands. The one you just entered will search from the cursor's position to the end of the file. The second command, `^K`, will search from the cursor's position to the start of the file. After you enter the search command, this dialog will appear:

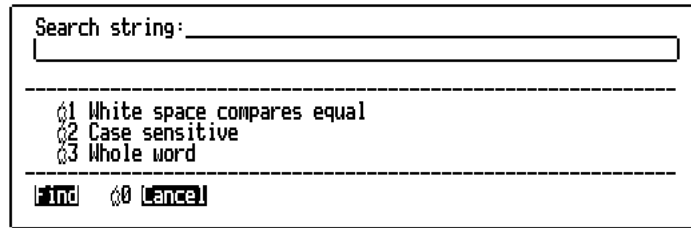


Figure 3.1

The cursor will be in the search string box; go ahead and enter the string you want to search for. Should you make a mistake entering the search string, there are some line editing commands you can use to correct the mistake. Just about all of the character-oriented commands, such as the right and left arrow keys, are active, while the line-oriented commands, such as scroll down, are not. Once you are satisfied with the search string, press **RETURN**. The editor will search for the first occurrence of type. If the editor cannot locate a search string, it will respond by beeping the speaker. If the editor does locate the search string, the cursor will be positioned on the first character of the target string. To continue searching the file for the same string, use the editor macro **⌘L** (or **OPTIONL**).

There are several options available with the string search command. You can select and deselect these options using the **⌘** key followed by a number from 1 to 3; a check mark will appear beside the option in the dialog when the option is selected. The first option treats all sequences of spaces and tabs as if they were a single space, both in the search string and in the text file. This is useful when you want to search for a pair of fields, like `lda` and `#2`, but you don't care if the fields are separated by spaces, tabs, or some combination of these characters. The case sensitive option makes the editor case sensitive, so that the search string `LDA` would not find `lda`, for example. The whole word option will only find a string if it is surrounded by characters other than alphabetic characters, numbers, or the underscore character. This is used to find variables that may be common characters or sequences of characters. For example, one of the strings the editor found when you searched for type was `Types` in the file name `Types.Rez`. If you ask the editor to search for whole words, it will skip this string.

Search With Replacement

This feature is an extension to the search feature. The command for search down with replacement is ⌘J. The command for search up with replacement is ⌘H. This time, you need to enter two strings, a search string and a replace string, so there are two edit line items in the dialog.

Search string: _____

Replace string: _____

1 White space compares equal
 2 Case sensitive
 3 Whole word
 4 Replace all

Replace ⌘0 Cancel

Figure 3.2

Once you enter the search string, use the tab key to move to the next field, and enter the replace string. Try entering `type` for the search string, and your name for the replace string, so that you can see how replacement works. The same commands for editing the replacement string as mentioned in search will also work here. Pressing the return key starts a sequence where the editor finds a string, and shows this dialog:

Replace with "target"?

Replace ⌘1 Skip ⌘2 Cancel

Figure 3.3

As the dialog suggests, pressing the return key replaces the string, and ⌘1 does not change the string. In either case, the editor will move on to the next occurrence of the string, stopping if there are no more. Selecting Cancel stops the process.

There is a new option in the search and replace dialog, too, called Replace All. If you select this option, all of the occurrences of the search string are replaced with the replace string. This is done throughout the file, not just from the cursor position down (or up). In a large file, this could take some time, so you will see a spinner at the bottom right of the screen as the command does its work. Pressing the ⌘. will stop the process.

Cut, Copy, Paste and Clear

These features can be used separately or combined. For example, if you have typed a section of program in your file, but you have typed the program block in the wrong place, you can use the cut commands to first cut (or remove) the block from the file. Then paste (or insert) the removed block at the desired location. As a second example, suppose you had a block of code which was to be used in several locations throughout the program. You could type every duplicate portion, or

you could use the copy feature to create a copy of the duplicate block, then use the paste command to insert the duplicate block wherever it was needed.

Cut

When the cut command is issued, the editor will go into select mode automatically. There are two different select modes which can be toggled: line-oriented and character-oriented. Line-oriented select means that only whole lines can be selected. Character-oriented select means that you can select any lines or any portion of the lines. When you are editing an assembly language program, you will be in line-oriented select automatically. Setting and using character-oriented select is covered in Chapter 13.

Let's try out the cut command. Enter the cut command, `␣X`. The editor goes into select (line-oriented) mode. You may use any cursor positioning commands we have talked about so far to move the cursor. Notice that as the cursor moves, the lines which the cursor moves over will be highlighted in reverse-video. These are the selected lines. Go ahead and select a few lines. When you have finished selecting the text, press `RETURN`. The selected information is then copied into a file on the work prefix called `SYSTEMP`. The selected text is waiting in the `SYSTEMP` file, and can be put back into the file using the paste command (covered later in this section) or left as is. If you want to cancel the cut, press `ESC` instead of `RETURN`.

You can also use the mouse to select text. Move the mouse to one position in the file, then drag the mouse to another position. Once the text is selected, you can use `␣X` to cut the text. Normally, when you use the mouse, you select individual characters. You can also select a word at a time by double-clicking to start the selection, or select lines by triple clicking when you start the selection.

Copy

The copy feature is similar to cut. Instead of removing the selected text, the editor will make a copy of the selected text and put it in the `SYSTEMP` file. Go ahead and enter the the copy command, `␣C`. The editor is in select mode. Select a few lines and press `RETURN`. The editor has placed a copy of the selected text in the `SYSTEMP` file. Notice that the use of the copy command will replace the selected text from the cut we just performed. If you want to cancel the copy, press `ESC` instead of `RETURN`.

As with the Cut command, you can, of course, use the mouse to select the text first, then use `␣C` to copy the text.

Paste

This feature is used to enter the text selected with the cut and copy commands back into our file. Try this command out. Position the cursor some place in the file where it will be obvious that the selected text will be added, then press the paste command, `␣V`. The selected lines will then be inserted above the cursor.

Clear

The clear command, also called the delete command, is used to delete selected text. `⌘DELETE` starts the selection mode. After selecting text and pressing the return key, all of the selected text is deleted.

Closing The File

When you are finished editing your text file, the command to exit the editor is `⌘Q`. You will see a prompt that asks if you want to save the file, choose No so the changes are not saved.

Normally, before leaving a file, you would use the `⌘S` command to save the file first. If you forget, the prompt you just saw reminds you that you are about to lose the changes you have made, and gives you a chance to recover.

For More Information...

This chapter was intended to show you some of the basic features of the editor, but there are many more features you may want to learn about as you become more proficient with ORCA. Chapter 13 covers the editor in more detail, discussing topics like using the mouse in dialogs, editing multiple files, and customizing the editor.

Chapter 4

The Fundamental Assembler Directives

This chapter is a guided tour through the most basic assembly language directives. Upon completion of this chapter, you should be able to write simple, but complete, assembly language programs. Because this is a tutorial, there are many programming examples for you to try. Topics covered in this chapter are:

- Assembly language statements.
- Comment lines.
- Various assembler directives.
- Global labels.
- Data segments.

Introduction

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. We will look at the ten 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 cover in this chapter are listed below.

ANOP	Assembler no-op.
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	Declare global constant.
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 statements 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 machine language instruction \$60. The way that these instructions are coded is very standard, with almost every 65816 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 statement 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 ADD4 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.

The format for a statement, using the JSR instruction as an example, is shown below.

<u>label</u>	<u>op code</u>	<u>operand</u>	<u>comment</u>
label	jsr	sub1	call sub1

Figure 4.1 Statement Fields

By default, ORCA is case insensitive, which means that a lowercase letter like a is treated exactly like its uppercase equivalent, A. The only exception is in strings of characters. Throughout this manual, we will conform to the most common coding practice of writing programs using mostly lowercase letters, but in text, uppercase letters will be used for operation codes so you can scan the text for the quicker, and not confuse them with words. For example, this convention makes it easier to distinguish at a glance the difference between the assembly language instruction AND and the word and. As a general rule of thumb, global labels will start with a capital letter. This is simply a handy convention favored by the author; you can ignore it if you like.

The ORCA assembler can be made case sensitive, but this is generally more of a pain than it is worth. The main reason you can make the assembler case sensitive is for writing subroutines that will be called by the C language. For more information, see CASE and OBJCASE, two directives described in Chapter 19.

Label

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

Chapter 4: The Fundamental Assembler Directives

through Z), numeric (0 through 9) or the underscore character or tilde. A label can range in length from 1 to 255 characters. Each character is significant. 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 operation code. ORCA is normally case insensitive. This means you can use a lowercase letter anywhere that an upper case letter is used, and they mean the same thing.

Op code

The operation code, or op code, is required with 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.

Operand

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 65816. ORCA formatting of instruction set operands is covered in Chapter 18. 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 Chapter 18 if you would like to know more.

Comment

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 affect 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.

There is only one case in the ORCA assembler where the operand field is optional, and that is with the directives that start a code segment, namely START, DATA, PRIVATE and PRIVDATA. When you code these directives, it is critical that, if you use a comment and no operand, the comment start in column 41 or beyond.

The examples below illustrate the main points of an assembly language statement.

	lda	label+1	load accumulator from data
!			stored at first byte
!			after LABEL
	ds	length*400	reserve LENGTH * 400 bytes
	ds	(length+1)*400	
L1	equ	10/14+1	define the constant L1
n1	gequ	-1	define the global constant N1

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 affect 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 asterisk (*), semicolon (;) or the exclamation (!) in the first column is also treated as a comment; any keyboard character can be used after the first character. Finally, a line with a period (.) in column one is a special kind of comment line called a sequence symbol. Sequence symbols are not normally 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 to place a space after the . character.

```
! This is a comment line.  
; This is, too.  
* You can also make a comment line in this manner.  
. This comment line uses the sequence symbol.
```

Before moving on, it is worth noting that the conditional assembly language supports variables. These variables are coded like a label, but begin with the & character. Whenever a variable appears in a source statement – even in a comment – it is expanded. For that reason, you should avoid putting something that the assembler might mistake for a variable in your comments. In a nutshell, this means that you should not follow the & character with an alphabetic character, tilde (~), or underscore (_) unless you want a previously defined variable to be expanded.

Directives

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 GS/OS path name; if the path name includes spaces, it must be included in quotes. The current prefix is assumed when only a partial path name is specified (see Chapter 5 for more information on path names). 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. As an example, suppose the source code for a program is in a file named MyProg.asm, and you want your executable program to be named :MyFile:MyProg. Then the first line in your program would be:

```
keep    :MyFile:MyProg
```

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

Chapter 4: The Fundamental Assembler Directives

named code segments. The `START` directive requires a label. The label on the `START` directive becomes the name of the code segment. You can also use an optional operand, which defines a load segment. This important concept will be covered in more detail in Chapter 6. The `END` directive has no operand, and usually no label. It is not possible to branch to a line containing an `END` directive. 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
          phk
          plb

lab2      lda    lab1
          lda    #0
          rtl
          end

Seg2      start
lab2      lda    lab2
lab1      lda    lab1
          rts
          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. It is 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. Think of a segment the same way you would think of a procedure, function, or subroutine in a high-level language. 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` directives 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 with the `EQU` and `GEQU` directives, and are called equates.

To define a constant, place the name of the constant in the label field, the `EQU` or `GEQU` in the op code field, and the value of the constant in the operand field. Of course, it always helps to comment. For example,

User's Manual

```
one      equ    1          define one
two      equ    1+1        define two
four     equ    two*two    define four
```

As shown, you can use expressions in the operand, and you can use constants defined by earlier equates. It is also possible to use expressions that reference labels, the current location counter, and externally defined constants.

Another use of EQU is to represent the value of the current location counter, as in:

```
here     equ    *
```

EQUs can also be used to identify a variable within a group of data. (That is, to simulate high-level language records in assembly language.) In the example below, FOUR will be the address of the fourth value past the label ONE, since the first floating-point number stored at ONE begins zero bytes beyond ONE, and each value requires four bytes of storage.

```
four     equ    *(3*4)
one      dc     f'1,2,3,4,5,6,7,8,9,10'
```

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 direct page address or long address, but it is better to get in the habit of defining before use to avoid problems later.

ANOP

The ANOP directive is used to assign a label to the current location counter. The EQU directive can also be used for this, but use of ANOP is more efficient. ANOP is often used to build tables or blocks of common data, as in:

```
table    anop                table of subroutine addresses
          dc     a4'SUB1'
          dc     a4'SUB2'
          dc     a4'SUB3'
tableEnd anop
```

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. By specifying a label with the DC directive, you can assign a variable name to the data. In this section, we will only talk about characters, integers, and hexadecimal values. If you need to enter one of the other types, see Chapter 18.

Chapter 4: The Fundamental Assembler Directives

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
E	extended precision floating point
F	single precision floating point
H	hexadecimal
I	integer
R	hard reference
S	soft reference

Table 4.1 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 65816 likes its addresses, and also the way the ORCA math libraries and the Apple IIGS toolbox like to find numbers. The I format by itself defaults to a two-byte number.

```
count    dc    i'1,1+1,3,4'
bigNum   dc    i8'1000000000000'
array1   dc    50i'1'
```

The second example demonstrates that integers come in several lengths. If you use an I for the type specifier, without coding a length, 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 in the range between -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 numbers include the digits 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, so 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.

User's Manual

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'
```

While hexadecimal DS statements are a great way to put specific bytes into memory, you don't have to use them to code numeric values as hexadecimal numbers, and in fact it is easier and clearer if you do not. For example, if you want to place the value \$ABCD in memory as a integer (with the bytes reversed) you would have to code the statement as

```
dc      h'CD AB'
```

reversing the bytes manually. On the other hand, the assembler recognizes hexadecimal numbers just as easily as it recognizes decimal numbers, so you can get exactly the same effect and be clearer in the process with this statement:

```
dc      i'$ABCD'
```

The last example we will look at shows how you can mix data types on a single line. For a very reasonable example, let's assume that you want to place a carriage return code at the end of a line of characters. Since the carriage return code is represented in ASCII as \$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.

```
errMsg  dc      c'Error! ',h'0D'

return   equ     $D
errMsg   dc      c'Error! ',l'return'
```

As mentioned earlier, there are several more data types supported for special uses. 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 peripheral devices and for the subroutine libraries, but if you are writing your own subroutines to output directly to the Apple text screen, you may want to have the most significant bit set. The MSB directive, described Chapter 19 of the reference manual, gives you a way to do that.

Before moving on, let's 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. Examples are:

```
num      equ     10
space    ds      100          allocate 100 bytes of memory
         ds      num*2        allocate 10 words of memory
```

Global Labels

So far, we have mainly talked about local labels, where a label in one code segment is "invisible" to 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.

Chapter 4: The Fundamental Assembler Directives

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. 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 direct page or long address 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. Its label receives the value of the current location counter. Its most common use is to define an alternate entry point into a subroutine.

```
Sub1      start
          jsr    Sub2          execute subrtn; value returned on
!                                     stack
          pla
          beq    s1            Based on value, enter Sub3 at
!                                     beginning or at alternate entry
!                                     point
          jsr    Sub3_a
          bra    s2
s1         jsr    Sub3
          lda    #1            Indicate Sub3 executed
s2         lda    #0            Indicate Sub3_a executed
          rts
          end

Sub3      start
size      equ    3

          clc
          adc    #size

Sub3_a    entry
          asl

          rts
          end
```

DATA Segments

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 begin 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.

User's Manual

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.

```
                keep    rdWrt
                mcopy   rdWrt.macros
*****
*
*   Example program showing data segments
*
*****
*
Main            start
                using   Common            gain access to data segment

                phk
                plb                        ensure code & data in same bank

                lda     #80                store max size of line in 1st byte
                sta     buffer            of line buffer

                jsr     ReadLn             read in a line of text
                puts    buffer            call macro to print the line just read

                lda     #0                tell the shell there are no errors
                rtl
                end                    return to the shell

*****
*
*   Data segment for program
*
*****
*
Common          data
return          equ     $0D                ASCII code for carriage return
char            ds      2                input character buffer
buffer          ds      82               output buffer
                end

*****
*
*   ReadLn routine
*
*****
*
ReadLn          start
                using   Common            gain access to data segment

                short   I,M                use 8-bit accumulator, X, Y registers
                ldy     #1                init. index into line buffer
```

Chapter 4: The Fundamental Assembler Directives

Top	phy		save Y register- destroyed by GETC macro
	getc	char	call macro to read char from keyboard
	ply		restore Y register
	iny		increment line buffer index
	lda	char	
	sta	buffer,Y	store char into line buffer
	cmp	#return	check end-of-line
	beq	Out	
	cpy	#81	don't go past end of buffer
	blt	Top	
Out	dey		Y is 1 too big
	sty	buffer+1	store line size in 2nd byte of buffer
	long	I,M	reset accumulator, X, Y to 16 bits
	rts		
	end		

Chapter 5

Advanced Shell Features

With the basics behind us, we can start looking at some of the advanced features of the shell. You should keep in mind that even in this chapter we will not look at all of the capabilities of the ORCA shell. After getting comfortable with the material in this chapter, you should plan on skimming Chapter 12 to look for topics which interest you. Topics covered in this section are:

- File naming conventions.
- Redirecting input and output.
- Pipelines.
- The program development process.

File Naming Conventions

What Is A File?

A file is a logical collection of bytes internal to the computer or stored on a disk. Files are used to contain the programs you write, the facilities used on the computer (i.e. the shell), and data created or used by these programs and facilities. There are several types of files used by ORCA, such as text (or source), object, library, and executable. The type of file is a clue to the file's purpose. The source file contains the program you type in. An object file contains the result of assembling your program. Library files contain library subroutines. The executable files contain the program code, which is ready to be executed.

File Names

A file name is attached to a file. File names are necessary to distinguish between files. Files used by ORCA use the same naming conventions as GS/OS. While other disk formats may someday be supported, the most commonly used disk format as this manual is written is the one first used by ProDOS, and that is the one that will be described here. Under the ProDOS File System Translator (FST), file name can be up to fifteen characters long. These characters are composed of the letters(A..Z), the digits (0..9) and periods (.). The first character of a file name must be a letter. GS/OS is not case sensitive, but the most recent versions do preserve the case you use when you create files, and this version of the ORCA shell supports that practice. Examples of legal file names are:

Exercise
Problem8
A.OUT

Path Names

The path name of a file is a form of "road map" which allows ORCA to locate a file. A path name is composed of directory names, followed by a file name. The directories and file name are separated by colons, with no intervening spaces. Each directory in the path name is a subdirectory of the preceding directory, and is also a root directory of the succeeding directory. A full path name begins with the volume name; a partial path name (a path name not beginning with a volume name) will use the current directory as the root of the given path. For example, the full path name

```
:HardDisk:Example:MyFile
```

would tell ORCA that the file named MYFILE is found in the directory named EXAMPLE on the volume named HARDISK.

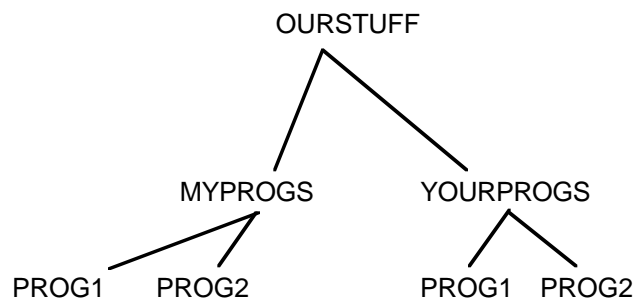
While GS/OS uses the colon as a separator character, ProDOS and many other file systems use the slash character. GS/OS supports the use of the slash character as a separator character when you enter a path name, so long as you use all slashes. GS/OS always returns path names using the colon character for a separator. The shell carries this one step further, allowing you to use any single typeable keyboard character as a separator by setting the {Separator} shell variable.

For example, you can type either of these commands, and they will do exactly the same thing:

```
edit 13:rinclud:types.rez
edit 13/rinclud/types.rez
```

Directory Walking

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



Assume that the current prefix is OURSTUFF:MYPROGS. If you want to access PROG1 in the YOURPROGS directory, you can use the partial path

```
..:YOURPROGS:PROG1
```

to get to it.

Device Names

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

```
show units
```

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

Units Currently On Line:

Number	Device	Name
.D1	.APPLESCSI.HD01.00	:harddisk
.D2	.APPLEDISK3.5A	:ORCA
.D4	.AFP1	:AppleTalk
.D20	.CONSOLE	<Character Device>
.D21	.APPLESCSI.HD01.01	:harddisk2
.D22	.PRINTER	<Character Device>
.D23	.NULL	<Character Device>
.D24	.DEV2	:Ram
.D25	.DEV3	<Character Device>

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

```
catalog .d1
```

will have the same effect as

```
catalog :harddisk
```

Standard Prefixes

ORCA provides prefixes which can be substituted for path names. You can obtain a listing of the standard prefixes for your system by typing the command

```
show prefix
```

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

User's Manual

System Prefix:

Number	Name
*	:harddisk:
@	:harddisk:
8	:ORCA:
9	:harddisk:
10	.CONSOLE:
11	.CONSOLE:
12	.CONSOLE:
13	:ORCA:libraries:
14	:ORCA:
15	:ORCA:Shell:
16	:ORCA:Languages:
17	:ORCA:Utilities:
18	:ORCA:

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

```
14:MyProg
```

and it would have the same effect as

```
:ORCA:MyProg
```

There are a total of 32 numbered prefixes supported by GS/OS, numbered 0 to 31. The first eight of these are no longer used, and their predefined meanings have been replaced under GS/OS. Prefixes 8 to 17 have predefined meanings under GS/OS, the Apple IIGS tools, or the ORCA shell, but you can use any of the prefixes from 18 to 31 for your own purposes. For example, if you kept your programs in a directory called MYSTUFF, you could rename prefix 18 to correspond to :ORCA:MYSTUFF using the command:

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

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

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

you can use the path name

```
18:myprog
```

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

Chapter 5: Advanced Shell Features

- * The asterisk indicates the boot prefix. The boot prefix is the name of the disk that GS/OS executed from.
- @ The @ prefix is the home prefix for people using networked computers.
- 0-7 These prefixes were used under ProDOS, and are still supported by GS/OS so old programs will work. They should not be used under GS/OS or the ORCA shell. In particular, no program should use prefixes 0 to 7 and any prefix from 8 to 31, since some of these prefixes are used for the same purpose under the two operating systems, and using both sets can cause conflicts. For example, the current, or default, prefix is prefix 0 under ProDOS, and prefix 8 under GS/OS.
- 8 This is the GS/OS default prefix. It is used whenever a partial path name is specified.
- 9 Prefix 9 is the program's prefix. Whenever a program is executed, prefix 9 is set to the directory where the program was found.
- 10 Prefix 10 holds the name of the standard input device or file. GS/OS sets this name to .CONSOLE by default; the .CONSOLE driver uses the keyboard for input.
- 11 Prefix 11 holds the name of the standard output device or file. GS/OS sets this name to .CONSOLE by default; the .CONSOLE driver uses the text screen for output.
- 12 Prefix 12 holds the name of the error output device or file. GS/OS sets this name to .CONSOLE by default; the .CONSOLE driver uses the text screen for output.
- 13 Prefix 13 is the library prefix. The linker searches the library prefix for libraries when unresolved references occur in a program. The Rez compiler searches a folder called RInclude for resource description files. Many other compilers also search a folder in this prefix for special header files.
- 14 Prefix 14 is the work prefix. This is the location used by various programs when an intermediate work file is created. If a RAM disk is available, this prefix should point to it. (Standard prefixes are changed with the shell's PREFIX command.)
- 15 Prefix 15 is the shell prefix. The command processor looks here for the LOGIN file and command table (SYSCMND) at boot time. Later, it looks here for the editor, which in turn looks for its macro file (SYSEMAC), tab file (SYSTABS) and, if present, editor command table SYSECMD.
- 16 Prefix 16 is the languages prefix. The command processor looks here for the linker, assembler, and compilers.
- 17 Prefix 17 is the utilities prefix. When a utility is executed, the command processor looks here for the utility. Help files are contained in the subdirectory HELP of the UTILITIES directory.

Redirecting Input and Output

The Apple IIGS supports two character-output devices and one character-input device. When you call the Apple IIGS Text Tools to read a character or string, or when you use any of ORCA's GET macros, the input comes from the keyboard. Input redirection lets you tell ORCA to take the characters from a file instead. When you call the Apple IIGS Text Toolkit to write a character, you have a choice of two devices: standard output and standard error output. Normally, both send the characters on to the screen. ORCA lets you redirect these devices separately to a disk file, a printer, or any other device recognized by GS/OS.

For example, when you specify a HELP command, the output is printed on the screen. Using redirection, the output can be moved, or redirected to the printer. Go ahead and try a simple redirection:

```
help delete >.printer
```

Be sure a printer is connected – if not, the system will hang, waiting for a response from the printer. If the printed output seems strange, or your printer does not work, check Chapter 12 for details on how to configure the .PRINTER driver that comes with ORCA.

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

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

For example, suppose you have a program which accepts student test scores from the keyboard and generates classroom statistics. The name of the program is TESTSTATS, and the data file is SCORES. Additionally, you want the results to be printed, not displayed on the screen. You could use the following command to accomplish this:

```
teststats <scores >.printer
```

After printing the results, you might want to run the program again and add the results to the end of an archive file called STATHISTORY. You could use the command

```
teststats <scores >>stathistory
```

The Program Development Process

In Chapter 1, you put together a simple program, and then, using one shell command, assembled, linked, loaded, and executed the program. In this section, we will cover this process in much greater detail.

When you assemble a program, ORCA can execute many different processes, depending on the options you choose.

<u>commands</u>	<u>command description</u>
ASSEMBLE COMPILE	assemble your program
LINK	link your program
ASML CMPL	assemble and link your program
ASMLG CMPLG RUN	assemble, link, and run your program

Assembling A Program

The first program that is involved in preparing a program for execution is the assembler. The assembler takes the program you typed into the editor as input, and through a series of steps, creates object modules as output. Object modules are an intermediate form of your program which contain definitions of global symbols, references to symbols contained in other object modules, the machine code for the program, and enough information to combine the object module with other object modules and produce an executable program.

Let's assume you want to assemble a file called MYPROG. The simplest command you can use to do this is

```
assemble myprog
```


There are several optional parameters associated with the assemble command. If you want to create a listing of the source code, you could get one by typing +L right after ASSEMBLE. +S will list the symbol tables used in the program. These parameters can be used in any order. For example, to get both a source listing and symbol tables, you would enter the command:

```
assemble +l +s myprog
```

This command would assemble the source file named MYPROG, print a listing of the program to standard out, and print the symbol table.

One other flag will be used often enough that we will introduce it here. Normally, the assembler does not pause when it hits an error. This is fine for automated builds, or in a program that is far enough along in the development cycle that errors are rare, but you may want to stop and look at errors if you are sitting at the computer. To do this, use the +W (wait) flag. For example,

```
assemble +w myprog
```

will cause the assembler to stop when an error is found. Type  to abort immediately to the editor to fix the problem, or any other key to continue.

As your programs get longer, you will find that one of the most useful features of the assembler is its ability to do partial assemblies. Partial assemblies let you assemble individual segments from the program. If you make a change or find an error in one segment, it is simple to

correct the error in the bad segment and relink the program. Without partial assembly, the entire program will have to be assembled over. To illustrate, if it took five minutes to assemble a large program (about 20,000 lines) and one minute to link, then it takes six minutes to get your program ready to execute. If you found a flaw in the program, then you are faced with another assembly. To assemble only the segment which has been corrected will usually take less than one minute. The linking process will remain constant, about one minute. So the entire process of correcting the error takes less than two minutes, as opposed to six minutes, a time savings of over 67%!

In order to perform a partial assembly, you need a way to specify which object segments you want reassembled. When you are assembling a program consisting of two or more segments, the first segment, when assembled, will be placed in an object file having an extension .ROOT, and the remaining segments will be placed in an object file with an alphabetical extension. For example, if you created a program called MYPROG made up of three segments called SEG1, SEG2, and SEG3, then after assembling all three segments, SEG1 would be in MYPROG.ROOT, and SEG2 and SEG3 would be in MYPROG.A. Now suppose you made a change to SEG2. Would it make sense to reassemble all of the segments? Not really. Use the ASSEMBLE command with the NAMES parameter.

```
assemble myprog names=(seg2)
```

This command will only assemble SEG2. The result of this partial assembly is that there are now three files, MYPROG.ROOT, MYPROG.A, and MYPROG.B. MYPROG.ROOT and MYPROG.A are left undisturbed. MYPROG.B will contain the new version of SEG2. When the linker (covered in the next section) links all the segments together, it will look in MYPROG.ROOT first and then look for MYPROG.A. If the .A file is found then MYPROG.B is searched for. If a .B file is found, then the linker will search for MYPROG.C. In this case, a file MYPROG.C is not found so the linker will stop with MYPROG.B. The linker will get SEG2 from MYPROG.B. When the linker goes through MYPROG.A, the linker will know it found SEG2 from a "higher" object file, and will only accept SEG3 from MYPROG.A, ignoring the SEG2.

There are other parameters associated with the assembly commands that will not be mentioned now. See the section called "Command and Utility Reference" in Chapter 12 for more information.

Linking A Program

After your program is assembled, the next step is to link the program. The link editor (or linker) is a program which combines all specified object files and libraries, resolving all references, and creates an executable file. There are three commands (as well as some aliases for them) that will do a link. LINK does a link only, giving you lots of control, but at the cost of added complexity. ASML does an assembly first, then a link. RUN also does an assemble and link, but finishes up by executing the program. All of these commands can accept linker parameters. If you specify +L, the link editor will output a link map, which is a listing of the segments in the program. +S causes the symbol table to be printed. You can also specify other object files that are to be linked. Finally, you can specify the name of the executable file as a keep name. Note that using the combined assemble and link command, ASML, specifying the +L and +S parameters will generate listings and symbol tables for both the assemble and link editor steps.

Some examples of the link commands are:

```
asml myfile.asm keep=myfile

link +l +s myfile keep=myfile

link myfile routine1 routine2 mylibrary
```

The first example will assemble, then link, the source file MYFILE. The executable file will be called MYFILE.OUT. Example two will link the object file named MYFILE, list the link map and the global symbols, and will name the executable file MYEXEFILE. The third example will link three object files and a library. It should be noted that the third example will not create an output file, since the KEEP option was not used.

The last example points out that the linker can link several object files and library files together. Object files specified explicitly are always searched before the system libraries. Once all object files and library files specified in the command line are linked, if there are still unresolved references, the linker will search all library files in the library prefix to try and resolve the references. The process of building a final executable program from separately compiled/assembled source files is called separate compilation.

There is more information about the link editor in Chapter 14.

Executing a Program

The output of the link editor is an executable file. Generally, the executable file is relocatable. Relocatable, in computer jargon, means that the file doesn't have to be located at a certain address. The loader, a part of GS/OS, will take the executable file, find a location in memory large enough to hold it, and then load the file into that memory, patching it as necessary. Once the executable file is in memory, it is ready to run. If you used the assemble, link and go commands ASMLG or RUN, then the executable file is automatically loaded and executed. To execute a program that has already been assembled and linked, simply type the name of the executable file.

Chapter 6

Advanced Assembler Directives

We have already looked at the assembler directives that are essential 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. This chapter will look at the directives which are not used in macros, while Chapter 8 will cover the remainder. Topics to be covered in this chapter:

- COPY directive.
- Format control directives.
- Setting case sensitivity.
- Setting the most significant bit of character constants.
- Load segments.
- Changing the word size.
- Miscellaneous directives.

COPY and APPEND

The COPY directive tells the assembler to go assemble another file, and come back when the file has been assembled. The most common use for the COPY directive is to include a series of statements or subroutines in more than one program, or in several pieces of a program that are assembled separately. The COPY directive uses a GS/OS path name in the operand. A similar directive is APPEND, which does basically the same thing, but does not return to the original source file after it has been used.

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 several dozen levels deep.

For example, suppose you had a series of thirty EQU directives in a file named MYPROGS:CONST which defined some constants. You wanted to use these constants in several segments, but don't wish to continually retype the thirty EQU statements. You could insert the following statement in each segment to copy the file which contains the constant definitions:

```
copy MyProgs:Const
```

When the assembler reaches this statement, the MYPROGS:CONST file will be copied into the new segment. Of course, you should place the copy statement before you make use of the defined constants.

Format Control Directives

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

Let's start with the LIST directive, which can be used to turn the listing on or off. Like many of the format control directives, the operand for the list directive is either ON or OFF. The default for this directive is off. Shouldn't a listing be generated automatically? 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 for every program assembly. Another directive, SYMBOL, allows you to turn the printing of symbol tables on or off. It, too, takes an operand of ON or OFF.

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. The default for ERR is on. For example, to suppress the listing of error lines, use the directive

```
ERR OFF
```

You may have noticed that when the LIST ON directive is used, the assembler writes the code that it is generating at the left edge of the screen. You also may 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. The default value is off. For an example, if you wanted to see the expanded code, then you would place the directive shown below in your code.

```
EXPAND ON
```

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. The default is off. 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 printer to skip to the top of a new page. An example of the PRINTER and EJECT directives are:

```
printer on  
eject
```

By using the SETCOM directive, you can define the column where the assembler will no longer search a statement for operands. By default, this is column forty. You can specify a value

from 1 to 255 for the directive. For example, if you wanted to set the assembler so that comments start in column 35, use the directive:

```
setcom 35
```

The TITLE directive places a page number and title at the top of each page. 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. An example, using the TITLE directive is:

```
title 'My program, version #3'
```

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

<u>directive</u>	<u>directive use</u>
EJECT	new page on printer
ERR	list/don't list errors
EXPAND	expand DC directives
LIST	list/don't list source
PRINTER	send listing to printer
SETCOM	set comment column
SYMBOL	list/don't list symbol tables
TITLE	place title at top of each page

Setting Case Sensitivity

The ORCA assembler is normally case insensitive – that is, lowercase and uppercase characters are treated the same in label names. You can cause the assembler to be case sensitive. Case is specified as either case sensitive or case insensitive. Case sensitive means that the computer will find a difference between upper case (capitals) and lower case characters. Case insensitive means the opposite – the computer will see no difference between upper case and lower case characters. If you use the CASE directive, the assembler can be set case sensitive (ON) or case insensitive (OFF). For example, if you wanted your program to be case sensitive, use the following statement.

```
case on
```

Setting The Most Significant Bit In Characters

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 (set to zero), but in the case of the character screen display used by the Apple, it is on (set to one). 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.

Load Segments

Before examining the directives used to create load segments, let's take a moment to talk about what load segments are and why you might need to use them.

Under ProDOS 8 on the Apple II, and under many other operating systems, executable programs are actually memory images. That means that the program has been created to run at a specific memory location, and the file is simply a series of bytes that, when loaded at the ORG location, will execute. The same is not true on the Apple IIGS under GS/OS. On the Apple IIGS, executable files consist of one or more load segments. Each load segment has a header that gives overall information about it, like its name and how much memory it will require; a binary image, much like the BIN file format under ProDOS 8; and a relocation dictionary. The relocation dictionary allows the load segment to be placed anywhere in memory where there is enough free room.

As we mentioned, the load file can have more than one load segment. There are three main reasons for creating a program with more than one load segment. The first has to do with a hardware restriction of the 65816 CPU. While the 65816 can access memory from anywhere in a 16 megabyte range, and can jump to or do subroutine calls in that range, it cannot execute across a 64K boundary. Thus, while a program with two subroutines can have one subroutine in one 64K bank, and the second in another, neither subroutine can lay across a 64K boundary. If code were linked to run at a fixed memory location, this would not be a problem – the linker could simply insert enough bytes to cause a subroutine that would cross a boundary to be placed in the next 64K bank. Since code is relocatable, the linker cannot do this trick. The alternative, and the way things work on the Apple IIGS, is to limit each load segment to 64K. This means that if your program is longer than 64K, you will have to split it into more than one segment.

The second reason for segmenting your code has to do with the amount of memory on an Apple IIGS, and how that memory is used. If your program must run on a 256K machine, but it is a desktop application that uses QuickDraw II, the Window Manager, the Menu Manager, and so on, you will find that there isn't much memory left. In fact, using the tools we have as we write this manual, the largest free area of memory would be about 30K. That doesn't mean your program is limited to 30K, simply that that is the largest single piece available. If you break your program up into small segments, the loader can place your program in all of the small areas of memory that are available.

The last common reason to use a load segment is to create a custom direct page segment. This is an extremely useful feature, but it requires a little background to understand when and why such a load segment is needed. Later, in Chapter 9, we'll look at an example of this kind of load segment.

Now that you know at least three reasons for writing your program with more than one load segment, let's take a look at the directives that are used to do this. Actually, the directives are not new. To create a load segment, you simply place the name of the load segment in the operand field of the START, DATA, PRIVATE, or PRIVDATA directive that starts the segment. The linker will group each object segment with the same load segment name into the same load segment. All program segments without a load segment name will be placed in a special load segment called the blank segment.

As an example, let's write a program with three segments: a main code segment, a subroutine that initializes an array of memory, and a data segment that is 64K bytes long. Since the entire program is larger than 64K bytes, we must segment the code.

Chapter 6: Advanced Assembler Directives

```
                keep  stuff
*****
*
*   Sample multi-segment program
*
*****
Main           start

                phk
                plb
                jsr   Init
                lda   #0
                rtl
                end

*****
*
*   Data Area to be Initialized
*
*****
Common        data   LoadSeg1

array         ds      65536
                end

*****
*
*   Initialize the data area
*
*****
Init          Start
                using Common

                ldx   #0
                lda   #0
lbl           sta   >array,X
                inc   A
                inx
                inx
                bne   lbl
                rts
                end
```

Looking at the program, we see that there are two load segments. The first, which is the blank segment, consists of MAIN and INIT. The linker will collect these and place them in one load segment. The second load segment is LOADSEG1, which consists of the segment COMMON. Note that long addressing was used to access ARRAY (indicated by the '>' character), which is in COMMON. You must always use long addressing to access data in another load segment, even if they are small enough that the loader can put them into a single 64K bank. The

reason is simple: the fact that the load segments are small enough to fit into one bank is no guarantee that they will, in fact, be placed in the same bank by the loader.

Two directives allow close control of the error checking the assembler will do for you. Normally, the assembler checks to make sure that only long addresses are used to access data outside of the current segment. Advanced programmers will find cases where they can safely use absolute addressing. To disable the assembler's error checking, use the `CODECHK` and `DATACHK` directives.

The `KIND` directive can be used to specify the type of object segment you are creating. It is described in Chapter 19, and used in an example to create a direct page segment in Chapter 9.

Changing The Word Size

The 65816 is designed so that its registers can be used as either eight- or sixteen- bit registers. The CPU knows how large they are from looking at two bits in the processor status register. Bit 4 indicates what size the index registers (X and Y) are, and bit 5 gives the size of the accumulator. In both cases, a one indicates eight-bit operations, and a zero indicates sixteen-bit operations.

Normally, the assembler doesn't care how large the registers are. The exception is when immediate addressing is used. In that case, the assembler must know how large the registers are, so that it knows how many bytes to generate. Since the assembler has no idea what the setting of the processor status flags will be at run time, it is up to you to tell the assembler how large the registers are. You do this with the `LONGA` and `LONGI` directives. Both take an operand of either `ON` (sixteen-bit mode) or `OFF` (eight-bit mode). In both cases, the default is `ON`.

In practice, it is usually easier to use the `LONG` and `SHORT` macros from the macro library. These macros set the status register for the CPU and issue the appropriate `LONGA/LONGI` directives for the assembler at the same time, reducing the chance of making an error.

Miscellaneous Directives

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 and 6502

The 65816 in the Apple IIGS is really three CPUs in one. The story started with the 6502, used in the Apple II and many Apple //e computers. The 6502 is a small, eight-bit version of the 65816. The next member of the family is the 65C02, which has a few more instructions and addressing modes. The 6502 is a true subset of the 65C02: all instructions and addressing modes on the 6502 are also present on the 65C02. When the 65816 was designed, the same principle was followed. All instructions and addressing modes in the 65C02 are also present in the 65816.

Because of this arrangement, you can use ORCA to write programs for the 6502 and 65C02 by simply restricting yourself to eight-bit registers and not using any of the instructions and addressing modes not available on the chip you are writing for. The assembler can be of help when you do this. Two directives, 65C02 and 65816, will cause the assembler to flag an error if you use an instruction or addressing mode that is not legal for the chip selected. Both take ON or OFF for operands. With 65816 ON, all addressing modes and instructions are enabled. That is the default mode for the assembler. With 65816 OFF, the 65C02 directive controls whether the instructions and addressing modes missing on the 6502 are flagged as errors. The examples below show the correct setting for each chip.

65816 ON	65816
65816 OFF	65C02
65C02 ON	
65816 OFF	6502
65C02 OFF	

Positioning Code And The ORG Directive

The Apple II GS has a very sophisticated memory management system. Your program is converted, by the linker, into an executable load file. This file is made up of a binary image and relocation dictionary. This load file is placed into memory using the GS/OS loader. The binary image is patched to run at the load location using the relocation dictionary. All of this means that you do not have to tell the system where to load the program; the system will load your program wherever it finds enough free memory.

The ORG directive, a relic from the older Apple // machines, is really not needed on this system. It is used to force code to be located at a specified address. We recommend that you not use it at all. If you are writing ROM code, or writing programs for older Apple computers, see the full description in Chapter 19. Other assembler directives that might prove useful in writing ROM code include ALIGN, OBJ, and OBJEND. Chapter 19 also describes each of these directives.

Chapter 7

Using Macros

A macro is a series of statements, much like a program segment. When you make a call to a macro in your program, the entire series of statements is assembled. When you are programming, it makes sense to create a macro to perform a task that is used frequently (for example: the division of two numbers). This way, by inserting a call to the macro in your main program, you can "hide the ugly" and increase the readability of your programs.

In this chapter, we will learn how to use a macro in an assembly language program. Topics covered are:

- Macro use.
- The ORCA macro library.
- Keyword parameters.

Tools of the Trade

The ORCA assembler comes with over 100 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.

Let's start by getting used to 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 list in Table 7.1. The name of the macro is in the first column, a short description of what it is used for is in the second column. For a detailed description of any of these macros, see Chapters 22 to 23. These are only a few of the macros available. Although it is not strictly necessary, you may want to stop and read about these macros in the reference manual before we go on.

<u>macro</u>	<u>macro description</u>
GET4	read a four-byte integer
PUT4	write a four-byte integer
PUTS	write a string
PUTCR	write a carriage return
ADD4	add two four-byte integers
SUB4	subtract two four-byte integers
MUL4	multiply two four-byte integers
DIV4	divide two four-byte integers

Table 7.1 Macros For Our Example

User's Manual

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 Calc4.Asm. Don't try to assemble it right away!

If you are not familiar with the ORCA editor, typing in this program will be good practice, but if you prefer, it is only fair to point out that this program has already been typed in for you. The source for this program is on your samples disk in a folder called Text.Samples.

```
                keep    Calc4
                mcopy   Calc4.Macros
*****
*
*   Four-Byte Integer Calculator
*
*****
*
C4              start
                phk
                plb
;
;   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
                lda     #0
                rtl

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 four assembly language instructions appear in the entire program. All of the other lines of code are macro calls. Like all other statements, macro calls have a label field, an op code field, and an 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 got an error.

It is possible to tell the assembler to use the macro libraries that come with ORCA directly, but that is very inefficient. 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 calc4.asm
```

where CALC4.ASM 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 Calc4.Macros, the same name that was used in the MCOPY directive in the sample program. 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 directory. Macros are kept in the library directory, inside a folder called ORCAInclude. From your hard disk you would type 13:orcainclude:m=; from floppy disks you would have to substitute the actual location of the macros as you are using your system. After a bit of whirring, you will have a macro file.

Note that all of the parameters can be typed on the command line, if you like. For details about this, and more information on MACGEN in general, see Chapter 12. You can also use the on-line help facility by keying in

```
help macgen
```

Now you can assemble the program by typing

```
run calc4.asm
```

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. In addition to using the subroutine libraries, the macros call some of the toolkits built into the Apple IIGS. 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 eight 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 the free memory in your computer. If for some reason you need to use more than one macro file 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.

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 covers the operand and data formats, as well as the macros used to manipulate the data.

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 65816, 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.

The extras disk contains a subdirectory called Libraries:AIInclude. These are macros written by Apple Computer, Inc., for its Apple IIGS Programmer's Workshop product, and licensed by the Byte Works, Inc.

If you have installed ORCA on your hard disk using our installation procedures, you will find the AIInclude folder already in your libraries folder, just like the ORCAInclude folder.

Apple's tool macros, used to access the Apple IIGS toolbox, are definitely the standard, but for a variety of reasons, they are fairly simple. ORCA/M has an alternate set of tool macros that allow you to place the parameters passed to the tools on the same line as the name of the macro itself. The standard Apple macros use the name of the tool call as shown in the Apple IIGS Toolbox Reference preceded with an underscore, while the ORCA tool macros precede the name with a ~ character.

A quick example will give you an idea of just how much space and typing these macros can save. Here's a short section of code that uses Apple's macros to draw a white square on the screen:

```
ph2    #15
_SetSolidPenPat
ph2    #10
ph2    #10
_MoveTo
ph2    #10
ph2    #100
_LineTo
ph2    #100
ph2    #100
_LineTo
ph2    #100
ph2    #10
_LineTo
ph2    #10
ph2    #10
_LineTo
```

Here's an example using the ORCA tool macros. These macros create exactly the same program as the ones you just saw.

```
~SetSolidPenPat #15
~MoveTo #10,#10
~LineTo #10,#100
~LineTo #100,#100
~LineTo #100,#10
~LineTo #10,#10
```

You can see the ORCA macros is use in Chapter 9. For a comparison of both macros, see Chapter 21.

More About Macro 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 description starts with a formatting model of the macro. The number of parameters that are allowed can be found by counting the number of parameters in the format model. 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 by surrounding it in square ([]) brackets. 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 occupy a position in the macro's parameter list. Coding

```
put4    num1,cr=t,#4           wrong
```

would not right-justify NUM1 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
```

User's Manual

The easy way to remember this is to always put positional parameters last when you write a macro call in your program.

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 the macros provided with ORCA, and in fact, since there are so many, most 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 write assembly language source code.

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 in 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 below 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.

Let's 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 IIGS is supplied with several "toolkits," that is, a collection of routines written by Apple and accessed via macro calls. One of these, called `_WriteChar`, will print the character in the least significant byte of the two-byte value stored on the stack to the screen. To call the Text Tools, you need to load the X register with the toolkit call number and function number, push the value to be printed onto the stack, and then execute a long call to the tool dispatcher. You can hide the messy details in a macro named `_WriteChar`, shown below.

```
macro
_WriteChar
pha
ldx    #$180C
jsl    $E10000
mend
```

Enter this macro using the editor, just like you would enter a program. Afterwards, save it in a file called `STUFF.MACROS`, and then type the following program into a new file.

```

Main      keep    stuff
          mcopy   stuff.macros
          start
          phk
          plb

          ldy     #0           Initialize index to zero

top       lda     msg,Y        Load a character
          beq     quit         Check end condition
          phy     Save contents of Y register
          _WriteChar          Print a character
          ply     Restore Y value
          iny
          bra     top

quit      lda     #0
          rtl

msg       dc      c'Hello, World!',H'0D0A'
          dc      i'0'
          end
```

When you run this program, it will print the contents of `MSG` onto the screen. Note that the `PHA`, `LDX` and `JSL` instructions don't show up anywhere in the program we wrote – the assembler pulls them in from the `STUFF.MACROS` file. Also note how we are passing a parameter to the macro, in the accumulator.

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 in the label and operand fields. 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 anything at all, and the operand 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. One way to set the value of a symbolic parameter is by passing it as a parameter to a macro. Symbolic parameters defined in this way are always string variables. The string they contain is the label name that was coded in the operand field of 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 value that has been assigned to it.

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
mend

```

This macro is a little more complicated than our first one, 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 times by passing different operands and look at the source lines generated. 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.

```

list    on
mcopy   test.macros
gen      on

test    start
        phk
        plb
        add    I,J,K
lb1     add    K,J,I
        lda    #0
        rtl

I       dc     i'1'
J       dc     i'2'
K       dc     i'3'
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 ADC, instead of on the CLC, and see what happens. Using a label on a program line that contains a macro call does not define the label. It only gets defined if the macro places the label on a statement that it produces. You can also use the label in more than one place inside of the macro.

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 a really meaningful example until later, 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: &LAB, &NUM1, &NUM2, and &NUM3. 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 of the segment in which they were defined 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 is available to the second macro. For example, the code below will work (although it is meaningless):

```
macro
main
lcla    &num
newmac
sta     &num
mend

macro
newmac
lda     &num
mend
```


Symbolic parameters defined by the macro model line are also local.

There is also a way to define symbolic parameters so that they are global in scope. The following lines define three global symbolic parameters:

```
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 &STRINGS(4) in a statement would cause the fourth string in the array to be used. Arrays can have up to 255 elements. 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.

User's Manual

```
&num      seta    const-const:65536*65536
&num      seta    &num-1
&arr(4)   seta    16
&arr(&num) seta    &arr(4)
&logic    setb    &num>0
&logic    setb    1
&string   setc    '&string+'&num
&string   setc    10.5
&string   setc    'Here''s a quoted string'
```

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 lines that go into generating the code. If you need to look at those lines, you should make use of the TRACE directive, described in Chapter 20.

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, and 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 position of 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
```

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 somewhere to branch. Conditional assembly branch 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.

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 see how 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. It is named AIF. The operand is a logical expression followed by a comma and a sequence symbol. If the expression evaluates to zero, it is false, and if it evaluates to non-zero, it is true. The following code fragment produces four ASL A instructions. Again, you can try it out in a short sample program.

```
lcla &n
&n seta 4
.top
asl A
&n seta &n-1
aif &n>0,.top
```

Attributes

Occasionally, it is nice to know more about a label or symbolic parameter than what its value is. That is where attributes come in. Attributes give you a way of asking questions about a 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

User's Manual

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 whether 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 a macro model line symbolic parameter unless a call to the macro is made with an operand corresponding to a symbolic parameter. The new macro would look like this:

```

                                macro
&lab                           add     &num1, &num2, &num3
                                aif     C:&num3, .PAST
                                lcl     &num3
&num3                          setc    &num1
. past
&lab                           clc
                                lda     &num1
                                adc     &num2
                                sta     &num3
                                MEND
```

The next attribute 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    il '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.

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.

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 examples of how the conditional assembly language is used, look at any of the macros in the macro library.

Chapter 9

Writing Programs and Utilities

There are several different kinds of programs that run on the Apple IIGS. This chapter gives a quick overview of how to use ORCA/M to develop each of these kinds of programs.

The kind of program you probably run most often is called a system program. System programs have a file type of S16, and can be launched from program launchers like Apple's Finder or the ORCA shell. System programs are generally desktop programs with menu bars and windows, but you can also write text system programs, or programs that use the graphics screen without using menus and other features of the toolbox. In fact, the ORCA shell is a system program, but it is a text based application.

Another kind of program is the shell program, which has a file type of EXE. Shell programs cannot be executed from the Finder. A shell program is the easiest kind of program to write using ORCA, since you don't have to leave ORCA to run the program. You can create shell programs that use the graphics screen or the desktop environment. For example, the PRIZM desktop development program is a shell program that runs from the ORCA shell. Another example of a shell program is a utility, like the MACGEN utility you have used throughout this manual.

There are several specialized kinds of programs you can write that aren't covered here. Absolutely any kind of program can be written with ORCA/M, but for specialized programs you will need to find documentation covering the specialized aspects of the program you want to write. With the special information in hand, the techniques covered in this chapter can be applied to create the new program. This chapter ends with a list of some of the more common specialized programs, along with where to go to get more information.

Shell Text Programs

We'll start by looking at a text program called from the ORCA shell. While this may not be the kind of program you ultimately want to write, you should still know how text programs work from the ORCA shell. There are two reasons to take the time to learn about text shell programs. The first is that all of the rest of this chapter builds on this information, so you may miss valuable tips if you skip ahead. More importantly, with a few minor exceptions, desktop programs launched from the Finder make use of the same protocols we'll discuss here.

When you run a program from any program launcher, whether that is Apple's Finder, the ORCA shell, or some other environment, the program launcher calls the system loader to read the program from disk and place the program in memory. The system loader can also allocate direct page space for the program. The program launcher then sets up the registers and does a JSL to your program. Your program should only access memory that belongs to it, either because the memory is a part of the program and was loaded by the loader, or because you allocated more memory by calling the memory manager. When your program is finished, it returns to the calling program by using either a GS/OS Quit call or by doing an RTL, depending on the exact circumstances. We'll go over the exact conditions as we come to them.

When your program is called by the shell, it is called in native mode with both index registers and the accumulator size set to sixteen bits. The Tool Locator, Memory Manager, Loader,

Miscellaneous Tools and Text Tools have been initialized, and standard input, standard output, and error output are all set up to do reasonable things; the default setting for I/O is the CRT screen.

The registers contain several important parameters. In the accumulator is a user ID number, assigned by the loader when your program was loaded into memory. This user ID number should be used for any subsequent calls to any tools that require a user ID, most notably, the Memory Manager. The X and Y registers contain the most and least significant words of the address of the command line, respectively. The command line contains an eight-character shell identifier followed by the actual characters typed when your program was executed (with shell variables expanded and I/O redirection handled and removed). The shell identifier is useful if you are writing programs that will execute from more than one program launcher. Different program launchers may place different requirements on your programs; the shell identifier lets you check to see which shell you are running under, and therefore which requirements you must satisfy. The shell identifier used by both ORCA and APW (Apple IIGS Programmer's Workshop) is BYTEWRKS. The command line is null-terminated. Note that it is legal to pass zeroes in the X and Y registers, indicating that there is no command line or shell identifier. Most desktop program launchers, like Apple's Finder, pass zeros in the X and Y registers, since they do not have a command line to pass.

During execution, you may have reason to open disk files, initialize other toolkits, or reserve memory. It is your responsibility to make sure that all disk files are closed, all memory deallocated, and all toolkits shut down before you return to the shell.

One important point is that you should never reinitialize the Text toolkit. Any program launcher that is capable of executing an EXE file is required to initialize it for you. If you reinitialize the Text toolkit, you can defeat some of the features of the program launcher. Under ORCA, reinitializing the Text Toolkit can interfere with I/O redirection.

Different shells may initialize the Text Tools in different ways. Under ORCA, the Text Toolkit will be initialized with the Pascal protocol. If you are using the ORCA I/O macros, you will never even need to be aware of this, since all terminal control features are handled for you. If you are writing your own output routines, you should read up on the different protocols in the *Apple IIGS Toolbox Reference Manual*.

When you do return to the shell, the registers must be in sixteen-bit mode. The accumulator contains an error code used by the shell to determine if an EXEC file should stop execution. Normally, you will set the accumulator to zero before returning, indicating that there was no error. If there was an error, return the error number reported by the toolkit if it was a system error, or \$FFFF if it was an internal error detected by your program. You can exit with either an RTL or a GS/OS QUIT call. Note that some program launchers may require one or the other – ORCA supports both.

The sample program shown below illustrates these principles. It prints the user ID number, shell identifier and command line passed to it when it executes. It then loads the accumulator with a zero to indicate that no error has occurred and returns. Try running the program with a variety of things typed after the command name, especially input and output redirection. Be sure you understand what the output is, and why.


```

        keep  args
        mcopy args.macros
*****
*
*  Args - Echo command line arguments, shell identifier, user ID
*
*  Inputs:
*      A - User ID
*      X-Y - pointer to shell identifier and command line
*
*  Outputs:
*      A - error code; set to 0
*
*****
*
Args      start
cl         equ    0                      command line pointer

        phk                          use local data bank
        plb
        sta    userID                save the user ID
        sty    cl                    save the command line pointer
        stx    cl+2
        txa                          if cl pointer is null then
        ora    cl                    tell the user
        bne    lb1
        puts   #'Null command line pointer',cr=t
        brl    lb5
lb1        anop                      else
        puts   #'Shell ID: '        write the shell identifier
        ldy    #0
        ldx    #8
lb2        lda    [cl],Y
        phx
        phy
        sta    ch
        putc   ch
        ply
        plx
        iny
        dbne   x,lb2
        putcr
        puts   #'Command line: '    write the command line
        ldy    #8
lb3        lda    [cl],Y
        and    #$00FF
        beq    lb4
        phy
        sta    ch
        putc   ch
        ply
        iny
        bra    lb3
lb4        putcr

```

```

lb5      anop                      endif
         puts    #'User ID: '      write the user ID
         put2    userID,cr=t
         lda     #0                  return with no error
         rtl

userID   ds      2                  user ID
ch       ds      2                  current character
         end

```

Memory Management

Unlike ProDOS 8 programs, programs designed to run under GS/OS are almost always relocatable. In fact, the penalties for writing a program that is not relocatable are so severe that we will not even discuss fixed location programs in this chapter. Since programs are relocatable, there is no longer any need to present, or really to even be aware of, the memory map. In fact, since ORCA itself is relocatable, it isn't even possible to present a memory map.

Instead, you must be aware of the conventions used in the system for memory management. Basically, Apple has a memory manager built into the Apple IIGS ROMs, and all programs are expected to use it. Using memory that has not been reserved for your own use can lead to disastrous results.

When your program is executed, the system loader (a program built into GS/OS that loads and relocates programs) reserves enough memory for your program. If your program needs any memory outside of its physical bounds, it must call the Apple Memory Manager to get it. The Apple Memory Manager requires that you supply a user ID number. Except in very unusual circumstances, you should supply the user ID number assigned by the loader and passed to your program as a parameter.

Installing a New Utility

Once you have an executable file that runs under the ORCA shell, you may want to install it as a utility. The advantages of doing so are that the program can be executed from any directory without typing a full path name, and the utility shows up in the command table. Once it is in the command table, you can use right-arrow expansion to abbreviate the command, 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 :ORCA:Utilities prefix, but you may have moved it to your hard disk, if you are using one.
2. Add the program name to the command table. The command table is in the SHELL 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 or tab, type a U, which indicates that the command is a utility. An optional comment can be used to briefly describe the utility's purpose.

Be sure to put the command in the correct location. The order that commands appear in the command table determines how right-arrow expansion works. The shell expands the first command that matches all letters typed.

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

3. If you would like to have on-line help for the command, add a text file to the **HELP** directory in the **UTILITIES** prefix. The name of the help 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 **SHELL** prefix and call it **EDITOR**. When it is called by the system, the editor should do a **GetLInfo** call using the macro supplied with ORCA. Three 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 in 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. Finally, the **PARMS** field points to the error message to display on an error entry to the editor. The editor returns to the shell in the normal way.

To install an APW- or ORCA-compatible compiler into ORCA/M, see the section entitled "Installing a New Language" in Appendix C.

If you are writing a new compiler which runs in the ORCA environment, again, the **GetLInfo** call is used to fetch input parameters. This time, though, the **SetLInfo** call must be used just before returning to the shell to tell the shell what to do next. The section that describes these two shell calls explains what the compiler will see. They are described in Chapter 24. Your language will need a unique language number so the shell can associate source files for your language with your language processor. Apple Computer maintains a complete list of language numbers, and will assign a language number to you on request.

System Programs

Programs that are launched from the Finder have a file type of **S16**, and are called system programs. For the most part, system programs are written the same way **EXE** programs are written, but there are three additional requirements you need to keep in mind.

The two biggest differences between a system program (file type of **S16**) and a shell program (file type of **EXE**) other than the file type itself is that a system program must return to the program launcher with a **GS/OS Quit** command, and a system program must start and shut down all tools that it uses.

So far, we've created programs that use an **RTL** instruction to return to the shell. You can also use a **GS/OS quit** instruction, though. There are several parameters that you can use with the **GS/OS quit** call, but we'll stick to the simplest possibility here. If you would like more detail, refer to the *Apple IIGS GS/OS Reference*. Here's what you would use in place of the familiar **RTL** instruction to exit from a program:

	<code>_QuitGS qtRec</code>	return to the program launcher
<code>qtRec</code>	<code>dc i'0'</code>	Quit record

While it is a little more complicated, and takes a few more bytes of code, this actually works from the shell, as well as from system programs. From the shell, the effect is exactly the same as if you had used an RTL instruction, but you can't return to the Finder any way except by using this sort of Quit call.

A system program must start all tools for itself. A shell program, on the other hand, knows that the Tool Locator, Memory Manager, Loader, Miscellaneous Tools and Text Tools have been initialized. As it turns out, you can restart and shut down any of these tools except the Text Tools from inside of a shell program.

Putting all of these facts together, it turns out that most desktop programs can be written and debugged as EXE programs. Desktop programs don't generally use the text tools. The other tools can be started and shut down from within your desktop program, just as if the program were running from the Finder. When your program is finished, you leave the program with a GS/OS Quit call, again just as if you were running from the Finder. Since the shell supports all of these conventions, you can develop the program as an EXE file, which is the default file type created by the linker. There is a big advantage to you when you run an EXE file, as opposed to an S16 file, too. When the shell runs an S16 file, it has to shut itself down, then run the program. When your program finishes, the shell must be reloaded. All of that takes time – time you don't have to waste if the program is in an EXE file.

Once the program is finished, you do need to change the file type to S16 so the Finder will recognize the program and run it. You can change the file type on the program using the FILETYPE command, like this:

```
filetype myprog S16
```

Of course, the biggest chore to writing desktop programs is learning how to use the tools effectively to create menu bars, windows, and so forth. The details of how to use the tools are beyond the scope of this user's manual. There are many fine books that do cover the tools, though, and most of them give you examples using ORCA/M. Chapter 1 gave a brief overview of some of these books.

Other Kinds of Programs

You can create any kind of program with ORCA/M, but there are special requirements for most of these programs. The requirements generally fall into two broad categories: special headers, and different file types.

As one example, classic desk accessories have a file type of CDA, and must start with a special header. You can learn about Classic Desk Accessories by reading the appropriate chapter of the *Apple IIGS Toolbox Reference*. To set the file type, you would use the FILETYPE command, just as we did in the last section to create an S16 file.

The rest of this chapter talks about some of the most common types of programs other than shell programs and system programs, and tells you where to go to get more information. You can find sample code for some of these on your ORCA/M samples disk, and other sample code can be obtained from your local users group or from major online services.

The technical notes and file type notes listed as a source of more information are published by Apple Computer, Inc. These are released through APDA, on major online services, and are also available from many users groups.

Classic Desk Accessories

Classic Desk Accessories are the text programs you get access to when you hold down the open apple and control keys and press the escape key. Classic Desk Accessories are available from any Apple IIGS program that does not specifically go out of its way to disable them, although some Classic Desk Accessories can only be used from a particular operating system.

For more information about how to write Classic Desk Accessories, see Chapter 5 of the *Apple IIGS Toolbox Reference, Volume 1*. Additional information can be found in "Apple IIGS Technical Notes, #71: DA Tips and Techniques."

New Desk Accessories

New Desk Accessories are the programs you find listed under the Apple menu of most desktop programs.

For more information about how to write New Desk Accessories, see Chapter 5 of the *Apple IIGS Toolbox Reference, Volume 1*. Additional information can be found in "Apple IIGS Technical Notes, #71: DA Tips and Techniques."

Initialization Programs

Initialization programs are executed automatically as you boot your computer. ORCA/M comes with a couple of initialization programs that give you an idea of the range of things you can use them for. One sets up the .PRINTER driver, setting the defaults you select with either a Classic Desk Accessory or a Control Panel Device. The other is GSBug, the machine language debugger that you enter any time the Apple IIGS hits a break instruction.

For more information about how to write initialization programs, see "Apple II File Type Notes, File Type \$B6, ProDOS 16 or GS/OS Permanent Initialization File," and "Apple II File Type Notes, File Type \$B7, ProDOS 16 or GS/OS Temporary Initialization File."

Control Panel Devices

Control panel devices, or CDevs, are the programs that are executed by Apple's control panel. They are generally used for configuring something in your computer. One example is the .PRINTER driver initialization CDev included with ORCA/M 2.0.

For more information about how to write CDevs, see "Apple II File Type Notes, File Type \$C7, Control Panel New Desk Accessory Device (CDev)."

Chapter 10

Programming the Shell

Like many professional development systems, the ORCA shell has enough power that you can program the shell itself. This chapter introduces the commands and features that are used to program the shell. Topics covered are:

- Shell Variables.
- Passing Parameters to EXEC Files.
- Looping.
- Conditional Execution.

What Is an EXEC File?

In order to learn how to effectively program the shell, you must start by thinking about it in the right way. If you have never used a programmable shell, you probably think about a shell strictly in terms of the command line editor that executes a command when you type it. The shell certainly is that, but it is much more.

We start to see how powerful the shell can be when we use EXEC files. (EXEC files go by many names - on other systems, they may have been called shell scripts, command files, or a variety of other names. We use the name EXEC files to conform to Apple traditions.) In its simplest form, an EXEC file is simply a list of commands that could have been typed from the command line, but were instead saved in a file. Under ORCA, this file is given a special language stamp of EXEC. When you execute the file, the operating system recognizes the file as a list of shell commands, and executes those commands in turn.

So far, we have a genuinely useful facility to reduce repetitive tasks by letting us type them once, and save the commands. But if you could pass parameters to an EXEC file, and then test those parameters and conditionally execute statements based on the input, the facility would be even more powerful. Add the ability to define and set variables, and to loop over lists of strings, and things start to get very useful indeed. The shell can do all of these things, and more.

We have now come full circle. Knowing what lies ahead, you can start to understand why the way you approach the shell can have a big impact on how useful it is. The key concept for this chapter is to think of the shell not as a command processor, but as a language. EXEC files are the source files for the shell language. We will look at how variables are defined and used, what program control statements you have, and how to pass parameters.

Variables

To be useful, any language must have a way of holding, recalling and changing data. In traditional algorithmic languages like BASIC, C, Pascal and assembly language, this is done with variables. The shell is no exception. It, too, is an algorithmic language, and it uses variables to hold information. Shell variables are somewhat limited, though. They cannot be subscripted, and come in only one type: strings. This is a restriction, but since the primary use of the shell language is to deal with strings, setting up commands to be executed, the restriction is not as severe as it might seem at first.

There are basically three places variables come from. The first is a set of special variables recognized and changed by the shell. These provide certain status information, such as the name of the last command executed, and whether that command had an error. The second source is passed parameters. We will look more closely at passed parameters later, but for now the important point is that whenever the shell executes an EXEC file, it creates several variables whose values are used to pass parameters. Finally, you can define variables yourself.

The table below lists the names of the shell variables that have special meaning to the shell or to one of the utilities or languages in the ORCA development environment. It is very important that you avoid using variables with the names listed below, unless you understand what the variable is for. Not all of these are defined automatically – in some cases you must define them yourself. Once defined, however, the shell uses these variables for special purposes, and putting incorrect information into the variables can cause the shell to do some unexpected and unwanted things. Many of these variables will be discussed in the rest of this chapter. A few are used for advanced or specific tasks that we won't go into here. If the description of the variable sounds like something you want to use, see the reference section for details. In fact, the reference section gives a more detailed description of each of these shell variables.

{0}	The name of the EXEC file being executed.
{1}, {2}, ...	Parameters from the command line.
{#}	The number of parameters passed.
{AuxType}	Provides automatic auxiliary file type specification for the linker.
{CaseSensitive}	Controls whether shell variable values are case sensitive.
{Command}	The name of the last command executed.
{Echo}	Controls whether shell commands are echoed.
{Exit}	Controls how the shell handles errors in a script.
{Insert}	Controls whether the shell starts in insert or over strike mode.
{KeepName}	Sets up an output file name for languages.
{KeepType}	Sets the file type used by the linker for executable files.
{Libraries}	Libraries list for the linker.
{LinkName}	Sets up an output file name for the linker.
{Parameters}	The parameters of the last command executed.
{Prompt}	Sets the shell's prompt.
{Separator}	Sets the separator used when the shell prints path names.
{Status}	The error status returned by the last command or EXEC file executed.

Table 10.1: Reserved Variable Names

The SET Command

With the preliminary comments out of the way, it is time to look at how variables are actually defined and changed. This is done with the SET command.

```
SET [variable [string]]
```

The set command takes two parameters. The first is the name of the variable you want to change, and the second is the name of the string you want to set the variable to. The string starts with the first non-blank character past the name of the variable, and continues through the end of the line. If you try to set a variable that does not exist, the shell creates the variable. That avoids the need for a separate variable declaration statement. As an example, let's assume you are working in a tree structured directory, but accessing information from another directory. We will assume that the other directory is called

```
:HARDDISK:NEWPROJECT:SOURCECODE:MAINPIECE.
```

That's a lot to type every time you want to access a file! The problem can be solved by defining a variable called D:

```
SET D :HARDDISK:NEWPROJECT:SOURCECODE:MAINPIECE
```

Now that we have the variable defined, we need to know how to use it. To signal to the shell that a particular name is a variable that must be expanded, all you need to do is enclose it in soft brackets. For example, to catalog the directory, you would type

```
CATALOG {D}
```

The shell expands {D} to the full path name you set that variable to, then passes the line on to the command processor. The command processor never knows that you didn't simply type the line in with the full path name.

When the shell expands the variable, it doesn't insert any blanks. This lets you concatenate strings by simply placing them next to each other. For example, to copy the file MYFILE from your directory, type

```
COPY {D}:MYFILE
```

As you start to define your own variables, you will need to know what the restrictions are on their names, as well as how long strings can be. Shell variable names must be 255 characters long or less; if you exceed this limit, the shell automatically truncates the name. Variable names are case insensitive, so {MYVAR} and {MyVar} will expand to the same thing. The only characters that you cannot use in a variable name are the space, tab and the left and right soft bracket ({ }).

Strings can be up to 65535 characters long; again, they are truncated if you exceed the maximum allowed length. While it is not normally necessary, strings can be enclosed in quote marks. The only time this is normally necessary is when you are setting a variable to the null string, or when a string contains characters that have special meaning to the shell, like the semicolon (used to place more than one command on a line) or the greater than sign (used to redirect output). Some examples are shown below.

User's Manual

```
SET FOO ""
SET BAR "Double quotes "" inside quotes"
SET GORP "Some special characters: ; >a <b"
```

At the start of this section, when the syntax for the SET command was given, you may have noticed that the string and variable names are both optional. If you leave off the string, you might expect that the variable would be set to the null string. In fact, if you leave off the string, the shell prints the value of the variable. For example, after setting variables as we did above, if you typed SET BAR you would see the following:

```
SET BAR
Set BAR Double quotes " inside quotes
```

If you leave off the variable name, too, the shell lists all of the variables that are currently defined, along with their values. These features are mostly debugging aids – to print a string from an EXEC file, you would normally use the ECHO command.

The UNSET Command

Once a variable has been defined, you may want to get rid of it. This is done with the UNSET command. The unset command takes a variable name as its parameter, and deletes both the variable and its contents. If you have been following along by typing in the samples, you can clean up your system now with the following commands.

```
UNSET D
UNSET FOO
UNSET BAR
UNSET GORP
```

The Classic Start: Hello, World

One of the things we have the C language to thank for is the defacto standard for the first program you try in any new language. The program is very simple: it prints "Hello, world." to the screen. It may seem almost too simple, but the point is a good one: try to learn how to enter and execute programs before concentrating on the language. So let's give it a try.

At the start of this chapter, we said that the shell knows that a file is an EXEC file because the file's language stamp is EXEC. This opens up a whole new can of worms. ORCA uses source files instead of text files for one very important reason: each source file has a language number associated with it. ORCA uses the language number to decide which compiler, interpreter, or whatever, to call when a file is to be processed. If the language is EXEC, ORCA knows that the file is a command file, and that it is safe to execute the commands in the file when you type the name of the file.

Naturally, you need to know how to set the language of a file. To do that, type the name of the language. After that, when the editor creates a new file, it will be stamped with the language you specified.

Before writing our first shell program, we need to know how to write a string. The ECHO command is used for that purpose. The ECHO command starts with the first non-blank character, and writes all characters from there to the end of the line to standard out. The command we want to execute, then, is

```
ECHO Hello, World.
```

You can type this command from the command line, and like all shell commands, it will work.

Now let's put all of this together into a program. Follow the steps outlined below.

1. Enter the EXEC command to change the current language to EXEC:

```
EXEC
```

2. Enter the editor with a new file name. To call the program HI, type

```
EDIT HI
```

3. Enter the ECHO command into the new file, as it appears in the example above.

4. Save the file and exit the editor.

5. Type the name of the HI program. The shell responds by running it.

```
HI  
Hello, world.
```

Before we get off of the subject, there are a few more things about language names that you should be aware of. First, typing the language name simply sets the default language. The only thing this is used for is if the very next EDIT command creates a new file. If you edit an existing file, the language of the file you edit stays the same. Also, when you edit an existing file, the editor changes the default language to match the language of the file you edited.

If you happen to create a file which has the wrong language stamp, you can change the language stamp using the CHANGE command. For example, if you had failed to change the default language to EXEC before entering your program in the above example, the file would probably be stamped as ASM65816. To change it to an EXEC file, you would type

```
CHANGE HI EXEC
```

One final point that may not be obvious is that EXEC files, like any other program, can be installed as utilities. To do so, follow the same procedure outlined in the last chapter.

Passing Parameters

To get an idea of how parameters are passed, we will start off by changing the hello world program slightly, so that it prints all of the shell variables and values. To do that, edit the HI file and add this line to the end of the file:

```
SET
```

As you will recall, the SET command with no parameters lists all shell variables and their values. Now run the program with the command line shown below. You type what is in bold: the shell response is in normal type.

User's Manual

```
HI GORP STUFF "FOO BAR"
Hello, World!
Set Parameters GORP STUFF "FOO BAR"
Set 0 HI
Set 1 GORP
Set 2 STUFF
Set 3 FOO BAR
Set # 3
Set EXIT TRUE
Set Command SET
Set Status 0
```

You may see a few other shell variables mixed in with these, but the shell variables shown will all be in the list somewhere.

In this bewildering array of variables are three important groups. We will break these down and discuss each separately.

First, since this section is on passed parameters, let's look at the variables used to pass those parameters. Parameters are passed as a series of numbered variables, starting with {0}. {0} is the path name used to execute the EXEC file. The remainder of the command line, with I/O redirection removed, is contained in the variable {Parameters}. Thus, to duplicate the original command line from inside the EXEC file, you could use

```
{0} {Parameters}
```

The shell also breaks the parameters up into individual words. A word is any sequence of characters except blanks, with blanks used to separate the words. If you need to include a blank in a word, as we did above with FOO BAR, enclose the word in quotes. These words are passed in numbered parameters, starting with {1} and continuing on for each parameter. In the example above, there were three such parameters, {1}, {2} and {3}. Finally, the {#} variable is set to the number of parameters passed – in this case, 3. It is difficult to present meaningful examples of passed parameter use until we have looked at the statements used to control program flow, so we will put off an example until later.

Variables Defined by the Shell

Now that we have talked about passed parameters, there are four variables left in the list whose meaning may not be clear. Three of those are actually defined by the shell. These are {Status}, {Command} and {EXIT}. {Status} is used by the shell to return the status of the last command executed. It is always returned as a number. If it is zero, the last command executed with no error. If it is not zero, there was an error, and the number is the error number. Often, this is the error number reported by GS/OS or one of the Apple IIGS tools. If a command is returning some other kind of error, such as too many parameters, it will return a value of 65535 (\$FFFF). If the last command executed a program, {Status} is the value of the accumulator when the program returned to the shell. That is why all of the sample programs in this manual set the accumulator to zero before returning.

{Command} is the name of the last command executed. It is generally useful only in complex shell programs, where you might save its value in a shell variable for later examination. We won't deal with it here.

{EXIT} is a very important variable. If it is non-null, and if any command returns a {Status} that is non-zero, the shell will abort the EXEC file. This is a protective feature that prevents the shell from continuing to execute commands after an error has occurred. If you will be handling your own errors by checking {Status}, or if you want the shell to continue execution even after an error, you need to turn this feature off. This is done by including the command

```
UNSET EXIT
```

in your EXEC file. This must be done in the EXEC file itself, since the shell redefines the variable each time an EXEC file starts. Some of our later examples will do this.

Variable Scope

There may have been some other variables in the list of shell variables printed by your HI program. There is a lot more to these variables than meets the eye. To start to see why, we will rerun the HI EXEC file just created, but after setting a variable first. Type the following:

```
SET Canary Bird
HI GORP STUFF "FOO BAR"
Hello, World!
Set Parameters  GORP STUFF "FOO BAR"
Set 0  HI
Set 1  GORP
Set 2  STUFF
Set 3  FOO BAR
Set #   3
Set EXIT  TRUE
Set Command SET
Set Status 0
```

{Canary} didn't show up in the list of variables printed by the SET command. Now type SET from the command line. {Canary} is still there.

The reason is that variables have scope. A variable defined at the shell level is not automatically available in an EXEC file, and a variable defined in an EXEC file is not available in another EXEC file called by the first. You can, however, make the variable available. To do so, you use the EXPORT command.

```
EXPORT [variable1 variable2...]
```

Like the SET command, the EXPORT command lists all of the exportable variables if you do not include a variable name. If you include a variable, that variable is exported to any EXEC files called from the shell, or if EXPORT is used in an EXEC file, to any EXEC files called by that one. To see how this works, type

```
EXPORT Canary
HI STUFF GORP "FOO BAR"
Hello, World!
Set Canary Bird
Set Parameters  STUFF GORP "FOO BAR"
Set 0  HI
Set 1  STUFF
Set 2  GORP
```

User's Manual

```
Set 3 FOO BAR
Set # 3
Set EXIT TRUE
Set Command SET
Set Status 0
```

Now {Canary} shows up in the list of variables listed by the HI program.

So where did the other shell variables come from? In your LOGIN file, you can create shell variables using the SET command, like this:

```
set KeepName $
export KeepName
```

The LOGIN file is a special EXEC file. If an EXEC file called LOGIN is found in the system directory during the boot process, it is executed automatically. EXPORT commands are treated specially in the LOGIN file. When you use the EXPORT command from the LOGIN file, it is as if the variables were defined and exported from the shell level. This is not normally the case, as we shall now see. Go back into the HI program, and add these lines right before the SET command:

```
SET CANARY ANIMAL
SET ORCA WHALE
```

Now try executing the EXEC file, and then type SET from the shell. {CANARY} and {ORCA} work just fine from the EXEC file, but {ORCA} is gone and {Canary} is unchanged when you get back to the shell. This emphasizes that, with the exception of exported variables in the LOGIN file, variables defined in an EXEC file are not defined at the shell level, and changes to exported variables are changes to local copies only, and do not affect the original.

If that were the end of the story, there would be no way except the LOGIN file to set variables at the shell level, other than typing them in by hand. Of course, that is not the end of the story. The EXECUTE command is a special command designed to overcome this problem. EXECUTE is another way of executing an EXEC file. If you now type

```
EXECUTE HI STUFF GORP "FOO BAR"
```

and type SET from the command line, you will find that all variables and changes from HI have taken place at the shell level, including definition of passed parameters, and so on. EXECUTE makes variables defined at one level available to the variables at the next level. That is, if you run HI without the EXECUTE command, but call another EXEC file from within HI by issuing the EXECUTE command, then all of the variables known to HI will be available to the called EXEC file. Since HI was run from the command line without the EXECUTE command, HI's variables will not be known to the shell.

Loops

The first control statement we will look at is the for loop. The syntax for the for loop is

```
FOR variable [IN list]
[statement]*
END
```

The for loop is used to loop over a list of words. Like words on the command line, words in the for loop are separated by spaces. If a space is needed in a word, the word can be enclosed in quote marks. The for loop will loop over each of the statements between the FOR statement and the END statement, looping once for each word in *list*. The variable *variable* is set to each of the words in *list* successively as the loop executes. For example, let's assume you want to copy all of the dot files associated with the object file MYOBJ to a backup disk named :BACKUP - i.e., you want to copy MYOBJ.ROOT, MYOBJ.A, MYOBJ.B, and so on. Further, since we are in an EXEC file, you don't know how many dot files there are. This can be done with the EXEC file:

```
set exit on
set dest :backup:myobj
copy myobj.root {dest}.root
for dot in a b c d e f g h i j k l m n o p q r s t u v w x y z
    copy -c myobj.{dot} {dest}.{dot}
end
```

When the EXEC file executes, each of the files will be copied until an error occurs because the file to be copied does not exist, at which point the shell will exit the EXEC file.

If you leave the IN keyword and word list off of the FOR statement, the shell will loop over all of the input parameters, starting with parameter number one. We can use this feature to improve on this program by looping over several inputs, copying all of the dot files associated with each root file name listed. Since we must avoid exiting when the first file that does not exist is encountered, we will turn off error exiting with the UNSET EXIT command:

```
unset exit
set dest :backup
for file
    copy -c {file}.root {dest}:{file}.root
    for dot in a b c d e f g h i j k l m n o p q r s t u v w x y z
        copy -c {file}.{dot} {dest}:{file}.{dot}
    end
end
```

Assuming the EXEC file is called COPYDOT, then to copy all of the dot files associated with MYOBJ and HISOBJ, you could simply type

```
COPYDOT MYOBJ HISOBJ
```

The only bad thing about this shell program is that it will continue to try and copy dot files long after there are no more. For example, if your dot files stopped at MYOBJ.K, it will still loop over the copy commands for MYOBJ.L to MYOBJ.Z. This does no harm, but takes lots of extra time. In the next section, we will look at how to solve this problem.

Before leaving the section on looping, we need to mention the other loop statement, LOOP. Like the FOR statement, the LOOP statement loops over all of the statements between it and the matching END. Unlike FOR, it loops continuously until an error occurs or a CONTINUE statement is encountered. CONTINUE will be covered in the next section. Since FOR is far and away the most commonly used loop statement, we will not go into LOOP further here. For details, see Chapter 12.

Conditional Execution

The shell command language supports an if statement that works like the if statement in many high-level languages. Like FOR and LOOP, it ends with the END statement. You can also use ELSE and ELSE IF for more control. The structure looks like this:

```
IF condition
[statement]*
[ELSE IF condition
[statement]*]*
[ELSE
[statement]*]
END
```

where [statement]* is a list of statement(s) to be executed. As an example, let's correct the problem in our last sample program. To do this, we will also need the BREAK statement, which exits the closest nested FOR or LOOP statement. The corrected program looks like this, with changes in boldface:

```
unset exit
set dest :backup
for file
  copy -c {file}.root {dest}:{file}.root
  for dot in a b c d e f g h i j k l m n o p q r s t u v w x y z
    copy -c {file}.{dot} {dest}:{file}.{dot}
    if {status} != 0
      break
    end
  end
end
```

The new statements, shown in bold, will exit the inner loop when a COPY command returns an error.

In the example, the IF statement has a conditional expression. Conditional expressions can be made up of words, equal operators (==), not equal operators (!=) and parentheses. The equal and not equal operators are string comparisons. Normally, they are case insensitive; see the reference manual if you would like to perform case-sensitive compares. The only unusual thing is that all of the terms in an expression must be separated by spaces. For example,

```
IF ( {P} == {Q} ) == ( {P} != {R} )
```

is correct, but

```
IF ( {P}=={Q} )==( {P}!={R} )
```

would not work.

A moment ago, we looked at the BREAK command, which exited the closest nested loop. A similar command, the CONTINUE command, skips the remainder of the statements until the end of the loop is reached. For details, see Chapter 12.

With all of the basics out of the way, we will look at one fairly significant programming example. There are occasions when partial assemblies don't work well because the order of

subroutines is important. The CRUNCH utility, which recombines the object modules, helps, but it breaks up the ASSEMBLE and LINK steps. The following EXEC file does the job. It assumes that the program to assemble is called MYPROG, and the object file MYOBJ. Assuming the EXEC file is called BUILD, you would get a full assembly by typing BUILD. Type BUILD followed by a list of the subroutines to assemble, and a partial assembly is performed.

```
if {#} == 0
  asml myprog
else
  for parm
    set list {list} {parm}
  end
  assemble myprog names=({list})
  crunch myobj
  link myobj keep=myobj
end
```


Chapter 11

Introduction To The Shell

This chapter gives an overview of the entire ORCA programming environment (or shell). The chapters that follow give technical details about the ORCA system. Topics to be covered in this chapter are:

- The shell.
- The command processor.
- The text editor.
- The assembly process.
- Assembly language.

What is the Shell?

The shell is the interface between you and the ORCA System. Once ORCA is started, the shell program will remain active (resident) until you stop system execution. Using shell commands, you can execute the editor, compile a program, change the current language for the system, copy or delete files, execute a utility, and so on. The hierarchy of ORCA is shown in the diagram below.

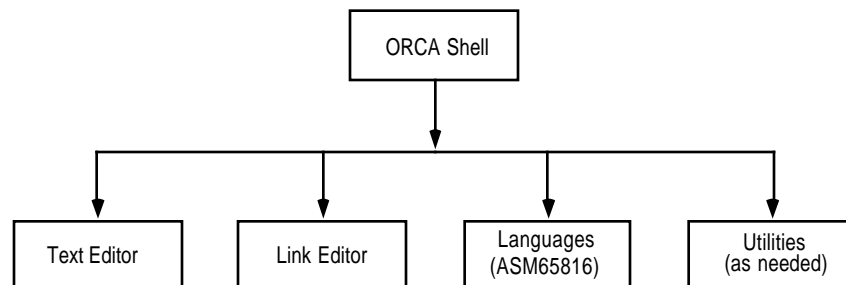


Figure 11.1 The ORCA System Hierarchy

The Command Processor

The command processor is the part of the shell that lets you type commands, then takes the commands you type and carries out some action. Features of the command processor include:

Shell Reference Manual

- A command interpreter for the interactive keyboard input of shell commands using an integrated line editor.
- Facilities for copying, renaming, deleting and moving files.
- Executable command files (EXEC files) for automatic execution of shell commands.
- Redirection of input and output.
- Pipelining of programs.
- The addition, deletion, and aliasing of shell commands.

These topics are covered in Chapter 12, "The Command Processor."

The Text Editor

ORCA provides a full-screen editor. Full cursor movement is allowed. There are over fifty commands used by the editor, and you can define up to twenty-six customized macros. Chapter 13, "The Text Editor," gives in-depth information on using the editor.

The Assembly Process

The shell provides complete facilities to convert your programs from source files to executable files loaded in memory. Advanced features include partial assembly and library modules. The process of changing an assembly language source file into an executable program starts with the assembler, which creates an object file by assembling the source file. The object file contains the program, along with the references to external labels, definitions of global labels defined in the source file, and enough information about the program to relocate it. The linker takes this object file, along with any other object files or libraries you ask it to use, and creates an executable file. All symbolic information has been resolved by this time, but relocation information is still included. Finally, when you execute the program, the loader (a part of GS/OS) finds a spot in memory large enough to hold your program, and places it there.

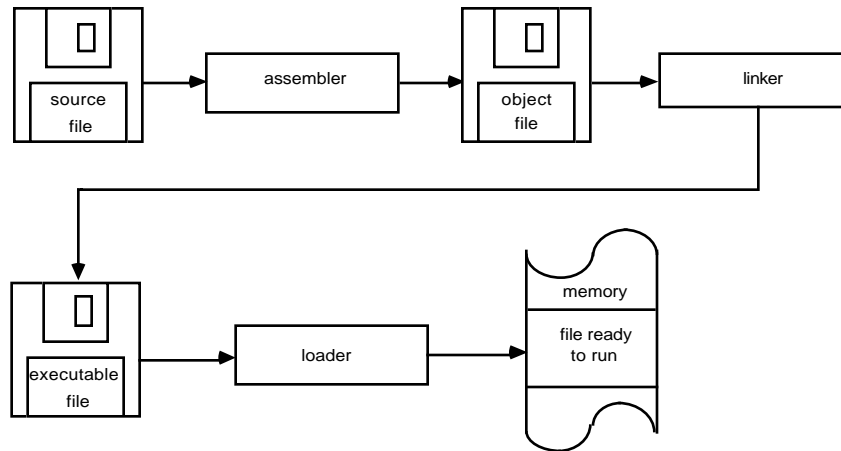


Figure 11.2 The Assembly Process

For more details, consult the following chapters.

- Chapter 12 "The Command Processor," describes the commands used.
- Chapter 14 "The Link Editor," provides detailed information on the linker.
- Chapter 15 "The Resource Compiler," gives details about an optional part of the development cycle, creating and using resources.
- Chapter 16 "GSBug Debugger," describes the machine language debugger GSBug, which you can use to help find problems in your programs.
- Chapter 17 "Running the Assembler," gives in depth information on assembling programs.

Assembly Language Programs

Assembly language is composed of assembly language instructions, assembler directives, and macros. During the assembly process, assembly language instructions convert directly to machine code, assembler directives instruct the assembler to perform some action, and macros are converted to instructions and/or directives. The following chapters can direct you to specific areas of the assembly language.

- Chapter 18 "Coding Instructions," covers the rules on using directives and macros and forming assembly language statements.
- Chapter 19 "Assembler Directives," describes the directives not specifically used to create macros.
- Chapter 20 "Macro and Conditional Assembly," covers macro construction.

Shell Reference Manual

Chapters 21 - 25 There are many useful macros contained in chapters 21 through 25, the macro libraries. These are macros used for math, system calls, GS/OS calls, and miscellaneous use.

Chapter 12

The Command Processor

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

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

The Line Editor

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

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

Command Name Expansion

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

~~CO~~RIGHT-ARROW

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

COMMANDS

Editing A Command On The Command Line

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

<u>command</u>	<u>command name and effect</u>
LEFT-ARROW	cursor left - The cursor will move to the left on the command line.
RIGHT-ARROW	cursor right - The cursor will move to the right. If the cursor is at the end of a sequence of characters which begin the first command on the line, the shell will try to expand the command.
␣ LEFT-ARROW	word left - The cursor will move to the start of the previous word. If the cursor is already on the first character of a word, it moves to the first character of the previous word.
␣ RIGHT-ARROW	word right - The cursor will move to the end of the current word. If the cursor is already on the last character in a word, it moves to the last character in the next word.
UP-ARROW or DOWN-ARROW	edit command - The up and down arrows are used to scroll through the 20 most recently executed commands. These commands can be executed again, or edited and executed.
␣> or ␣.	end of line - The cursor will move to the right-hand end of the command line.
␣< or ␣,	start of line - The cursor will move to the left-hand end of the command line.
DELETE	delete character left - Deletes the character to the left of the cursor, moving the cursor left.

<code>␣F</code> or <code>CTRLF</code>	delete character right - Deletes the character that the cursor is covering, moving characters from the right to fill in the vacated character position.
<code>␣Y</code> or <code>CTRLY</code>	delete to end of line - Deletes characters from the cursor to the the end of the line.
<code>␣E</code> or <code>CTRLE</code>	toggle insert mode - Allows characters to be inserted into the command line.
<code>␣Z</code> or <code>CTRLZ</code>	undo - Resets the command line to the starting string. If you are typing in a new command, this erases all characters. If you are editing an old command, this resets the command line to the original command string.
<code>ESCX</code> or <code>CLEAR</code> or <code>CTRLX</code>	clear command line - Removes all characters from the command line.
<code>RETURN</code> or <code>ENTER</code>	execute command - Issue a command to the shell, and append the command to the list of the most recent twenty commands.

Table 12.1 Line-Editing Commands

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

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

Multiple Commands

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

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

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

Scrolling Through Commands

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

Command Types

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

Built-in Commands

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

Utilities

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

The utilities themselves must all reside in the same subdirectory so that the command processor can locate them. The name of the utility is the same as the name of the command used to execute it; the utility itself is an EXE, or executable, file. Utilities are responsible for parsing all of the input line which appears after the command itself, except for input and output redirection. The command line is passed to a utility the same way it is passed to any other program. See Chapter 9 for step-by-step instructions on installing new utilities in ORCA.

Language Names

The last type of command is the language name. All source files are stamped with a language, which can be seen when the file is cataloged under ORCA. There is always a single system language active at any time when using ORCA. New files will normally be stamped as ASM65816.

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

Table 12.2 shows a partial list of the languages and language numbers that are currently assigned. CATALOG and HELP will automatically recognize a language if it is properly included in the command table. ProDOS has a special status: it is not truly a language, but indicates to the editor that the file should be saved as a standard GS/OS TXT file. Language numbers are used internally by the system, and are generally only important when adding languages to ORCA. They are assigned by Apple Computer, Inc.

<u>language</u>	<u>number</u>
PRODOS	0
TEXT	1
ASM6502	2
ASM65816	3
ORCA/PASCAL	5
EXEC	6
ORCA/C	7

Table 12.2 A Partial list of the Languages and Language Numbers

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

Program Names

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

Standard Prefixes

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

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

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

Shell Reference Manual

<u>Prefix Number</u>	<u>Use</u>	<u>Default</u>
@	User's folder	Boot prefix
*	Boot prefix	Boot prefix
8	Current prefix	Boot prefix
9	Application	Prefix of ORCA.SYS16
10	Standard Input	.CONSOLE
11	Standard Output	.CONSOLE
12	Error Output	.CONSOLE
13	ORCA library	9:LIBRARIES:
14	ORCA work	9:
15	ORCA shell	9:SHELL:
16	ORCA language	9:LANGUAGES:
17	ORCA utility	9:UTILITIES:

Table 12.3 Standard Prefixes

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

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

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

The @ prefix is useful when you are running ORCA from a network. If you are using ORCA from a hard disk or from floppy disks, prefix @ is set just like prefix 9, defaulting to the prefix when you have installed ORCA.SYS16. If you are using ORCA from a network, though, prefix @ is set to your network work folder.

The current prefix (also called the default prefix) is the one that is assumed when you use a partial path name. If you are using ORCA on a self-booting 3.5 inch disk, for example, prefix 8 and prefix 9 are both normally :ORCA:. If you boot your Apple IIGS from a 3.5-inch :ORCA disk, but run the ORCA.SYS16 file in the ORCA: subdirectory on a hard disk named HARDISK, prefix 8 would still be :ORCA: but prefix 9 would be :HARDISK:ORCA:.

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

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

The linker searches the files in the ORCA library prefix (prefix 13) to resolve any references not found in the program being linked. ORCA comes with a library file that supports the ORCA assembler macros; you can also create your own library files. For more information on creating and using ORCA assembler library files, see the discussion of the MAKELIB command in this chapter.

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

The work prefix (prefix 14) is used by some ORCA programs for temporary files. For example, when you pipeline two or more programs so that the output of one program becomes the input to the next, ORCA creates temporary files in the work prefix for the intermediate results (pipelines are described in the section "Pipelines" in this chapter). Commands that use the work

prefix operate faster if you set the work prefix to a RAM disk, since I/O is faster to and from memory than to and from a disk. If you have enough memory in your system to do so, use the Apple IIGS control panel to set up a RAM disk (be sure to leave at least 1024K bytes of memory for the system), then use the PREFIX command to change the work prefix. To change prefix 14 to a RAM disk named :RAM5, for example, use the following command:

```
PREFIX 14 :RAM5
```

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

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

```
EDITOR  
SYSTABS  
SYSEMAC  
SYSCMND  
LOGIN
```

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

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

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

Prefixes 0 to 7

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

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

Shell Reference Manual

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

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

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

File Names

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

<u>symbol</u>	<u>meaning</u>
.Dx	This indicates a device name formed by concatenating a device number and the characters '.D'. Use the command: SHOW UNITS to display current assignment of device numbers. Since device numbers can change dynamically with some kinds of devices (e.g. CD ROM drives) it is a good idea to check device numbers before using them.
.name	This indicates a device name. As with device numbers, the "show units" command can be used to display a current list of device names. The two most common device names that you will use are .CONSOLE and .PRINTER, although each device connected to your computer has a device name. .CONSOLE is the keyboard and display screen, while .PRINTER is a device added to GS/OS by the Byte Works to make it easy for text programs to use the printer.
x	Prefix number. One of the 32 numbered prefixes supported by GS/OS. See the previous section for a description of their use. You may use a prefix number in place of a volume name.

Chapter 12: The Command Processor

- .. When this is placed in a path name, it indicates that the reference is back (or up) one directory level.
- :
- This symbol, when inserted in a path name, refers to a directory. You can also use /, so long as you do not mix : characters and / characters in the same path name.

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

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

Here are some examples of legal path names:

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

Wildcards

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

```
DELETE MY=
```

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

```
DELETE MY?
```

would delete files that begin with MY after you responded yes to the prompt for each file. The wildcards can be used anywhere in the file name.

There are limitations on the use of wildcards. Some commands don't accept wildcards in the second file name. These commands are:

```
COPY
MOVE
RENAME
```

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

ASML
CMPL
CMPLG
COMPILE
DUMPOBJ

Types of Text Files

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

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

EXEC Files

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

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

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

could be typed into an EXEC file as

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

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

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

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

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

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

Passing Parameters Into EXEC Files

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

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

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

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

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

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

Programming EXEC Files

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

Variables

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

Shell Reference Manual

names are always enclosed in curly brackets ({ }), except when being defined in the SET, UNSET and FOR commands.

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

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

{0}	The name of the EXEC file being executed.
{1}, {2}, ...	Parameters from the command line. Parameters are numbered sequentially in the sequence in which they are entered.
{#}	The number of parameters passed.
{AuxType}	Provides automatic auxiliary file type specification. The variable contains a single value, specified as a hex or decimal integer. The AuxType string sets the auxiliary file type for the executable file produced by the linker. Any value from 0 to 65535 (\$FFFF) can be used.
{CaseSensitive}	If you set this variable to any non-null value, then string comparisons are case sensitive. The default value is null.
{Command}	The name of the last command executed, exactly as entered, excluding any command parameters. For example, if the command was :ORCA:MYPROG, then {Command} equals :ORCA:MYPROG; if the command was EXECUTE :ORCA:MYEXEC, then {Command} equals EXECUTE. The {Parameters} variable is set to the value of the entire parameters list.
{Echo}	If you set this variable to a non-null value, then commands within the EXEC file are printed to the screen before being executed. The default value for Echo is null (undefined).
{Exit}	If you set this variable to any non-null value, and if any command or nested EXEC file returns a non-zero error status, then execution of the EXEC file is terminated. The default value for {Exit} is non-null (it is the ASCII string true). Use the UNSET command to set {Exit} to a null value (that is, to delete its definition).

Chapter 12: The Command Processor

{Insert}	When you are using the shell's line editor, you start off in over strike mode. If the {Insert} shell variable is set to any value, the shell's line editor defaults to over strike mode.
{KeepName}	<p>Provides an automatic output file name for compilers and assemblers, avoiding the KEEP parameter on the command line and the KEEP directive in the language. If there is no keep name specified on the command line, and there is a non-null KeepName variable, the shell will build a keep name using this variable.</p> <p>This keep name will be applied to all object modules produced by an assembler or compiler. On the ASML, ASMLG and RUN commands, if no {LinkName} variable is used, the output name from the assemble or compile will also determine the name for the executable file. See {LinkName} for a way to override this.</p> <p>There are two special characters used in this variable that affect the automatic naming: % and \$. Using the % will cause the shell to substitute the source file name. Using \$ expands to the file name with the last extension removed (the last period (.) and trailing characters).</p>
{KeepType}	Provides automatic file type specification. The variable contains a single value, specified as a hex or decimal integer, or a three-letter GS/OS file type. The KeepType string sets the file type for the executable file produced by the linker. Legal file types are \$B3 to \$BF. Legal file descriptors are: EXE, S16, RTL, STR, NDA, LDA, TOL, etc.
{Libraries}	When the linker finishes linking all of the files you specify explicitly, it checks to see if there are any unresolved references in your program. If so, it searches various libraries to try and resolve the references. If this variable is not set, the linker will search all of the files in prefix 13 that have a file type of LIB. If this variable is set, the linker searches all of the files listed by this shell variable, and does not search the standard libraries folder.
{LinkName}	Provides an automatic output name for the executable file created by the link editor. The % and \$ metacharacters described for {KeepName} work with this variable, too. When an ASML, ASMLG or RUN command is used, this variable determines the name of the executable file, while {KeepName} specifies the object file name. This variable is also used to set the default file name for the LINK command.
{Parameters}	The parameters of the last command executed, exactly as entered, excluding the command name. For example, if the command was EXECUTE :ORCA:MYEXEC, then {Parameters} equals :ORCA:MYEXEC. The {Command} variable is set to the value of the command name.
{Prompt}	When the shell's command line editor is ready for a command line, it prints a # character as a prompt. If the {Prompt} shell variable is set to

any value except the # character, the shell will print the value of the {Prompt} shell variable instead of the # character. If the {Prompt} shell variable is set to #, the shell does not print a prompt at all.

{Separator} Under ProDOS, full path names started with the / character, and directories within path names were separated from each other, from volume names, and from file names by the / character. In GS/OS, both the / character and the : character can be used as a separator when you enter a path name, but the : character is universally used when writing a path name. If you set the Separator shell variable to a single character, that character will be used as a separator whenever the shell writes a path name. Note that, while many utilities make shell calls to print path names, not all do, and if the utility does not use the shell or check the {Separator} shell variable, the path names will not be consistent.

{Status} The error status returned by the last command or EXEC file executed. This variable is the ASCII character 0 (\$30) if the command completed successfully. For most commands, if an error occurred, the error value returned by the command is the ASCII string 65535 (representing the error code \$FFFF).

Logical Operators

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

<i>str1</i> == <i>str2</i>	String comparison: true if string <i>str1</i> and string <i>str2</i> are identical; false if not.
<i>str1</i> != <i>str2</i>	String comparison: false if string <i>str1</i> and string <i>str2</i> are identical; true if not.

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

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

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

Entering Comments

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

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

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

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

Redirecting Input and Output

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

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

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

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

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

```
ASSEMBLE MYPROG >.PRINTER
ASSEMBLE >.PRINTER MYPROG
```

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

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

Instead of using the keyboard for response to the AINPUT directive in a source file that is being assembled, the following example redirects a file (called ANSWERS) as input to the assembly process. Either one of the following commands would accomplish this.

```
ASSEMBLE <ANSWERS MYPROG
ASSEMBLE MYPROG <ANSWERS
```

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

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

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

<code>>&outputdevice</code>	Redirect error output to go to <i>outputdevice</i> .
<code>>>&outputdevice</code>	Append error output to the current contents of <i>outputdevice</i> .

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

The .PRINTER Driver

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

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

```
TYPE MyFile >.Printer
```

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

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

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

init called TextPrinterInit at boot time to configure the text printer driver, which is itself a GS/OS driver called TextPrinter.

Figure 12.4 shows the screens you will see when you use the CDev from Apple's Control panel or when you select the CDA from the CDA menu. The options that you can select are the same for both configuration programs; these are described in Table 12.5.

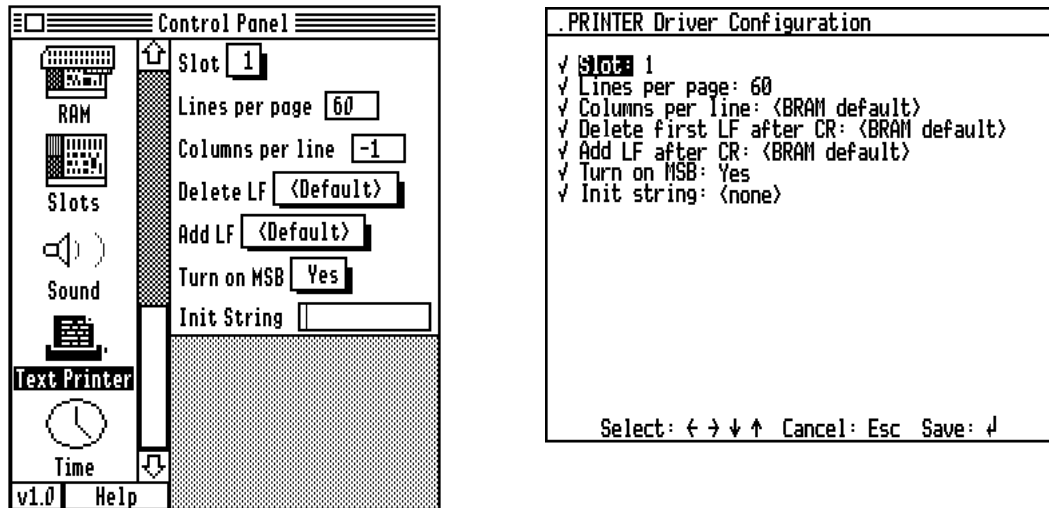


Figure 12.4: Text Printer Configuration Screens

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

- Delete LF** Some printers need a carriage-return line-feed character sequence to get to the start of a new line, while others only need a carriage-return. Some programs write a carriage-return line-feed combination, while others only write a carriage-return. This option lets you tell the printer driver to strip a line-feed character if it comes right after a carriage-return character, blocking extra line-feed characters coming in from programs that print both characters.
- You can select three options here: Yes, No, or BRAM Default. The Yes option strips extra line-feeds, while the No option does not. The BRAM Default option tells the printer driver to use whatever value is in the BRAM; this is the same value you would have selected using the printer configuration program in the control panel.
- Add LF** Some printers need a carriage-return line-feed character sequence to get to the start of a new line, while others only need a carriage-return. This option lets you tell the printer driver to add a line-feed character after any carriage-return character that is printed.
- You can select three options here: Yes, No, or BRAM Default. The Yes option adds a line-feeds, while the No option does not. The BRAM Default option tells the printer driver to use whatever value is in the BRAM; this is the same value you would have selected using the printer configuration program in the control panel.
- Turn on MSB** This line is a flag indicating whether the printer driver should set the most significant bit when writing characters to the printer. If this value is Yes the printer driver will set the most significant bit on all characters before sending the characters to the printer. If you code any number other than 0, the most significant bit will be cleared before the character is sent to the printer.
- Init string** This option sets a printer initialization string. This string is sent to the printer when the driver is used for the first time. With most printers and interface cards, there is some special code you can use to tell the printer that the characters that follow are special control codes. These codes are often used to control the character density, number of lines per page, font, and so forth. This initialization string, sent to the printer by the .PRINTER driver the first time the printer is used, is the traditional way of setting up your favorite defaults.
- You will find many cases when you will need to send a control character to the printer as part of this initialization string. To do that using the CDev configuration program precede the character with a ~ character. For example, an escape character is actually a control-[, so you could use ~[to send an escape character to the printer. The printer driver does not do any error checking when you use the ~ character, it simply subtracts \$40 from the ASCII code for the character that follows the ~ character, and sends the result to the printer. For example, g is not a control character, but ~g would still send a value, \$27, to the printer. From the CDA configuration program, just type the control character in the normal way; it will show up as an inverse character on the display.

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

Table 12.5: Text Printer Configuration Options

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

The .NULL Driver

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

Pipelines

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

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

```
PROG0 | PROG1 | PROG2
```

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

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

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

```
TYPE MYFILE | UPPER
```

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

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

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

The Command Table

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

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

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

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

As an example of what has been covered so far, the command table shipped with the system is shown in Table 12.6.

```

;
; ORCA Command Table
;
ALIAS          C      40      alias a command
ASM65816 *L    3       65816 assembler
ASML           C      1       assemble and link
ASMLG          C      2       assemble, link and execute
ASSEMBLE C     3       assemble
BREAK         C      25      break from loop
CAT            C      4       catalog
CATALOG       C      4       catalog

```

Chapter 12: The Command Processor

CC	*L	8	ORCA/C compiler
CHANGE	C	20	change language stamp
CMPL	C	1	compile and link
CMPLG	C	2	compile, link and execute
COMMANDS C	35		read command table
COMPACT	*U		compact OMF files
COMPILE	C	3	compile
COMPRESS C	32		compress/alphabetize directories
CONTINUE C	26		continue a loop
COPY	C	5	copy files/directories/disks
CREATE	C	6	create a subdirectory
CRUNCH	*U		combine object modules
DELETE	C	7	delete a file
DEREZ	*U		resource decompiler
DEVICES	C	48	Show Devices
DISABLE	C	8	disable file attributes
DISKCHECK	U		check integrity of ProDOS disks
DUMPOBJ	U		object module dumper
EDIT	*C	9	edit a file
ECHO	C	29	print from an exec file
ELSE	C	31	part of an IF statement
ENABLE	C	10	enable file attributes
END	C	23	end an IF, FOR, or LOOP
ENTAB	*U		entab utility
ERASE	C	44	Erase entire volume.
EXEC	L	6	EXEC language
EXECUTE	C	38	EXEC with changes to local variables
EXISTS	C	19	see if a file exists
EXIT	C	27	exit a loop
EXPORT	C	36	export a shell variable
EXPRESS	U		converts files to ExpressLoad format
FILETYPE C	21		change the type of a file
FOR	C	22	for loop
GDEBUG	U		application version of debugger
HELP	C	11	online help
HISTORY	C	39	display last 20 commands
HOME	C	43	clear the screen and home the cursor
IF	C	30	conditional branch
INIT	C	45	initialize disks
INPUT	C	13	read a value from the command line
LINK	*C	12	link
LINKER	*L	265	command line linker script
LOOP	C	24	loop statement
MACGEN	U		generate a macro file
MAKEBIN	U		convert load file to a binary file
MAKELIB	U		librarian
MOVE	C	34	move files
PASCAL	*L	5	Pascal compiler
PREFIX	C	14	set system prefix
PRIZM	U		desktop development system
PRODOS	L	0	ProDOS language
QUIT	C	15	exit from ORCA
RENAME	C	16	rename files
RESEQUAL *U			compares resource forks
REZ	*L	21	resource compiler
RUN	C	2	compile, link and execute
SET	C	28	set a variable
SHOW	C	17	show system attributes
SWITCH	C	33	switch order of files
SHUTDOWN C	47		shut down the computer
TEXT	L	1	Text file
TOUCH	C	46	Update date/time
TYPE	C	18	list a file to standard out
UNALIAS	C	41	delete an alias

UNSET	C	37	delete a shell variable
*	C	42	comment

Table 12.6 System Commands

Command And Utility Reference

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

UPPERCASE

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

italics

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

directory

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

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

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

GS/OS prefix numbers can be used for directory prefixes. An asterisk (*) can be used to indicate the boot disk. Two periods (..) can be used to indicate one subdirectory above the current subdirectory. If you use one of these substitutes for a prefix, do not precede it with a slash. For example, the HELP subdirectory on the ORCA disk can be entered as 6:HELP.

filename

This parameter indicates a file name, *not* including the prefix. The device names .CONSOLE and .PRINTER can be used as file names. Other character devices can also be used as file names, but a block device (like the name of a disk drive) cannot be used as a file name.

pathname

This parameter indicates a full path name, including the prefix and file name, or a partial path name, in which the current prefix is assumed. For example, if a file is named FILE in the subdirectory

DIRECTORY on the volume VOLUME, then the *pathname* parameter would be: :VOLUME:DIRECTORY:FILE. If the current prefix were :VOLUME:, then you could use DIRECTORY:FILE for *pathname*. A full path name (including the volume name) must begin with a slash (/) or colon (:); do *not* precede *pathname* with a slash if you are using a partial path name.

Character device names, like .CONSOLE and .PRINTER, can be used as file names; the device numbers .D1, .D2,Dn can be used for volume names; GS/OS device names can be used as volume names; and GS/OS prefix numbers, an asterisk (*), or double periods (..) can be used instead of a prefix.

- | A vertical bar indicates a choice. For example, +L|-L indicates that the command can be entered as either +L or as -L.
- A|B** An underlined choice is the default value.
- [] Parameters enclosed in square brackets are optional.
- ... Ellipses indicate that a parameter or sequence of parameters can be repeated as many times as you wish.

ALIAS

`ALIAS [name [string]]`

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

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

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

Now, to dump MYFILE in hexadecimal format, type

```
DUMP MYFILE
```

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

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

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

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

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

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

ASM65816

ASM65816

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

ASML

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

This internal command assembles (or compiles) and links a source file. The ORCA shell checks the language of the source file and calls the appropriate assembler or compiler. If the maximum error level returned by the assembler or compiler is less than or equal to the maximum allowed (0 unless you specify otherwise with the MERR directive or its equivalent in the source file), then the resulting object file is linked. The linker is described in Chapter 14. Assembler error levels are described in Appendix A.

You can use APPEND directives (or the equivalent) to tie together source files written in different computer languages; ORCA compilers and assemblers check the language type of each file and return control to the shell when a different language must be called. See the section “Assembling or Compiling a Program” in Chapter 2 for a description of the assembly and compilation process.

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

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

+D -D	+D causes debug code to be generated so that the source-level debugger may be used later when debugging the program. -D, the default, causes debug code to not be generated. The assembler ignores this flag.
-------	---

- +E|-E When a terminal error is encountered during an assembly from the command line, the assembler aborts and enters the editor with the cursor on the offending line, and the error message displayed in the editor's information bar. From an EXEC file, the default is to display the error message and return to the shell. The +E flag will cause the assembler to abort to the editor, while the -E flag causes the assembler to abort to the shell.

- +L|-L If you specify +L, the assembler or compiler generates a source listing; if you specify -L, the listing is not produced. The L parameter in this command overrides the LIST directive in the source file. +L will cause the linker to produce a link map.

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

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

- +S|-S If you specify +S, the linker produces an alphabetical listing of all global references in the object file; the assembler or compiler may also produce a symbol table. The ORCA Assembler, for example, produces an alphabetical listing of all local symbols following each END directive. If you specify -S, these symbol tables are not produced. The S parameter in this command overrides the SYMBOL directive in the source file.

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

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

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

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

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

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

Note the following about the KEEP parameter:

- If you use neither the KEEP parameter, the {KeepName} variable, nor the KEEP directive, then the object files are not saved at all. In this case, the link cannot be performed, because there is no object file to link.
- The file name you specify in *outfile* must not be over 10 characters long. This is because the extension .ROOT is appended to the name, and GS/OS does not allow file names longer than 15 characters.

NAMES=(*seg1 seg2 ...*) This parameter causes the assembler or compiler to perform a partial assembly or compile; the operands *seg1*, *seg2*, ... specify the names of the segments to be assembled or compiled. Separate the segment names with one or more spaces. The ORCA Linker automatically selects the latest version of each segment when the program is linked.

The object file created when you use the NAMES parameter contains only the specified segments. In ORCA Assembly Language, you assign names to segments with START, DATA, PRIVATE, or PRIVDATA directives. In most high-level languages, each subroutine becomes an object segment and the segment name is the same as the subroutine name.

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

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

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

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

ASMLG

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

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

ASSEMBLE

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

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

BREAK

BREAK

This command is used to terminate a FOR or LOOP statement. The next statement executed will be the one immediately after the END statement on the closest nested FOR or LOOP statement. For example, the EXEC file

```
FOR I IN 1 2 3
  FOR J IN 2 3
    IF {I} == {J}
      BREAK
    END
    ECHO {I}
  END
END
```

would print

```
1
1
3
```

to the screen. This order results from the fact that **BREAK** exits from the closest loop, the **FOR** **IN** 2 3, not from all loops.

CAT

```
CAT [-A] [-D] [-H] [-L] [-N] [-P] [-T]
    [directory1 [directory2 ...]]
```

```
CAT [-A] [-D] [-H] [-L] [-N] [-P] [-T] [pathname1 [pathname2 ...]]
```

This internal command is an alternate name for **CATALOG**.

CATALOG

```
CATALOG [-A] [-D] [-H] [-L] [-N] [-P] [-T]
    [directory1 [directory2 ...]]
```

```
CATALOG [-A] [-D] [-H] [-L] [-N] [-P] [-T]
    [pathname1 [pathname2 ...]]
```

This internal command lists to standard output the directory of the volume or subdirectory you specify. More than one directory or subdirectory can be listed to get more than one catalog from a single command.

- A GS/OS supports a status bit called the invisible bit. Finder droppings files, for example, are normally flagged as invisible so they won't clutter directory listings. The **CATALOG** command does not normally display invisible files when you catalog a directory; if you use the **-A** flag, the **CATALOG** command will display invisible files.
- D If the **-D** flag is used, this command does a recursive catalog of directories, showing not only the directory name, but the contents of the directory, and the contents of directories contained within the directory.
- H When this flag is used, the **CATALOG** command does not print the header, which shows the path being cataloged, or the trailer, which shows statistics about disk use.
- L The standard format for a directory listing is a table, with one line per file entry. When this flag is used, the **CATALOG** command shows a great deal more information about each file, but the information is shown using several lines.
- N This flag causes the **CATALOG** command to show only the name of the file, omitting all other information. Files are formatted with multiple file names per line, placing the file names on tab stops at 16 character boundaries. The resulting table is considerably easier to scan when looking for a specific file.

- P** The name of a file is normally displayed as a simple file name. Use of the **-P** flag causes the files to be listed as full path names. This option does make the file names fairly long, so the default tabular format may become cumbersome. Using this option with **-L** or **-N** clears up the problem.
- T** Most file types have a standard 3-letter identifier that is displayed by the catalog command. For example, an ASCII file has a 3-letter code of **TXT**. These 3-letter codes are displayed by the **CATALOG** command. If you use the **-T** flag, the **CATALOG** command displays the hexadecimal file type instead of the 3-letter file type code.
- This flag also controls the auxiliary file type field, which is shown as a language name for **SRC** files. When the **-T** flag is used, this field, too, is shown as a hexadecimal value for all file types.
- directory* The path name or partial path name of the volume, directory, or subdirectory for which you want a directory listing. If the prefix is omitted, then the contents of the current directory are listed.
- pathname* The full path name or partial path name (including the file name) of the file for which you want directory information. You can use wildcard characters in the file name to obtain information about only specific files.

```
:ORCA.DISASM:=
```

Name	Type	Blocks	Modified	Created	Access	Subtype
Desktop.DISASM	S16+	230	14 Aug 90	21 May 90	DNBWR	\$DB03
DISASM	EXE	101	15 Aug 90	15 Aug 90	DNBWR	\$0100
DISASM.Config	\$5A+	2	17 May 90	30 Apr 90	DNBWR	\$800A
DISASM.Data	TXT	95	10 Aug 90	20 Oct 88	DNBWR	\$0000
DISASM.Scripts	SRC	94	23 May 90	15 Aug 89	DNBWR	\$0116
Help	DIR	1	18 Sep 89	14 Sep 89	DNBWR	\$0000
Samples	DIR	1	13 Aug 90	14 Sep 89	DNBWR	\$0000
Icons	DIR	1	17 Sep 89	14 Sep 89	DNBWR	\$0000
Blocks Free:	1026	Blocks used:	574	Total Blocks:	1600	

Table 12.7 Sample CATALOG Listing

Table 12.7 shows the output from cataloging the ORCA/Disassembler 1.2 disk. This particular disk has a good variety of file types and so forth; we'll use it to see what the **CATALOG** command can tell us about a disk.

The first line shows the path being cataloged; in this case, we are cataloging all files on the disk **ORCA.DISASM**. The last line gives more information about the disk, including the number of blocks that are not used, the number that are used, and the total number of blocks on the disk. For ProDOS format disks, a block is 512 bytes, so this disk is an 800K disk.

Between these two lines is the information about the files on the disk. The first column is the file name. If the file name is too long to fit in the space available, the rest of the information will appear on the line below.

Next is the type of the file. Most file types have a three letter code associated with them, like **S16** (System 16) for a file that can be executed from the Finder or the ORCA shell, and **DIR**

(directory) for a folder. There is no three letter code for a file with a type of \$5A, so this file type is shown as the hexadecimal number for the file type. If a file is an extended file (i.e., if it has a resource fork), the file type is followed by a + character.

The column labeled "Blocks" shows the number of blocks occupied by the file on the disk. GS/OS is clever about the way it stores files, not using a physical disk block for a file that contains only zeros, for example, and programs are not necessarily loaded all at once, so this block size does not necessarily correspond to the amount of memory that will be needed to load a file or run a program; it only tells how much space is required on the disk.

The columns labeled "Modified" and "Created" give the date and time when the file was last changed and when the file was originally created, respectively. In this example, the time fields have been artificially set to 00:00 (something the Byte Works does for all of its distribution disks). When the time is set to 00:00, it is not shown.

The column labeled Access shows the values of six flags that control whether a file can be deleted (D), renamed (N), whether it has been backed up since the last time it was modified (B), whether it can be written to (W) or read from (R), and whether it is invisible (I). In all cases, if the condition is true, the flag is shown as an uppercase letter, and if the condition is false, the flag is not shown at all.

The last column, labeled "Subtype", shows the auxiliary file type for the file. For most files, this is shown as a four-digit hexadecimal number, but for SRC files you will see the name of the language.

```

Name           : Desktop.DISASM
Storage Type   : 5
File Type      : S16          $B3
Aux Type       : $DB03
Access         : DNBWR       $E3
Mod Date       : 14 Aug 90
Create Date    : 21 May 90
Blocks Used    : 139
Data EOF       : $00011A6B
Res. Blocks    : 91
Res. EOF       : $0000B215

```

Table 12.8

The tabular form used by the CATALOG command to show information about files is compact, but doesn't provide enough room to show all of the information about a file that is available from GS/OS. When the -L flag is used, the CATALOG command uses an expanded form to show more information about the file. Table 12.8 shows the expanded information for the Desktop.DISASM file. The name, file type, auxiliary file type, access, modification date and creation date fields are the same as before, although the order has changed and the fields that have a hexadecimal equivalent are shown using both forms. The old block count field has been expended, showing the number of blocks used by the data fork (the Blocks Used field) and the resource fork (labeled Res. Blocks) as two separate values. In addition, the true size of the file in bytes is shown, again split between the data fork and resource fork, as the Data EOF field and the Res. EOF field. Finally, the internal storage type used by GS/OS is listed.

For a more complete and technical description of the various information returned by the CATALOG command, see *Apple IIGS GS/OS Reference*, Volume 1.

CHANGE

`CHANGE [-P] pathname language`

This internal command changes the language type of an existing file.

-P When wildcards are used, a list of the files changed is written to standard out. The -P flag suppresses this progress information.

pathname The full path name or partial path name (including the file name) of the source file whose language type you wish to change. You can use wildcard characters in the file name.

language The language type to which you wish to change this file.

In ORCA, each source or text file is assigned the current default language type when it is created. When you assemble or compile the file, ORCA checks the language type to determine which assembler, compiler, linker, or text formatter to call. Use the CATALOG command to see the language type currently assigned to a file. Use the CHANGE command to change the language type of any of the languages listed by the SHOW LANGUAGES command.

You can use the CHANGE command to correct the ORCA language type of a file if the editor was set to the wrong language type when you created the file, for example. Another use of the CHANGE command is to assign the correct ORCA language type to an ASCII text file (GS/OS file type \$04) created with another editor.

CMPL

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

This internal command compiles (or assembles) and links a source file. Its function and parameters are identical to those of the ASML command.

CMPLG

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

This internal command compiles (or assembles), links, and runs a source file. Its function is identical to that of the ASMLG command. See the ASML command for a description of the parameters.

COMMANDS

COMMANDS *pathname*

This internal command causes ORCA to read a command table, resetting all the commands to those in the new command table.

pathname The full path name or partial path name (including the file name) of the file containing the command table.

When you load ORCA, it reads the command-table file named SYSCMND in prefix 15. You can use the COMMANDS command to read in a custom command table at any time. Command tables are described in the section “Command Types and the Command Table” in this chapter.

The COMMANDS command has one other useful side effect. Any restartable programs that have been loaded and left in memory will be purged, thus freeing a great deal of memory.

COMPACT

COMPACT *infile* [-O *outfile*] [-P] [-R] [-S]

This external command converts a load file from an uncompact form to a compacted form.

infile Input load file. Any OMF format file is acceptable, but the only files that benefit from the COMPACT utility are the executable files, such as EXE and S16.

-O *outfile* By default, the input file is replaced with the compacted version of the same file. If you supply an output file name with this option, the file is written to *outfile*.

-P When the -P flag is used, copyright and progress information is written to standard out.

-R The -R option marks any segment named ~globals or ~arrays as a reload segment. It also forces the bank size of the ~globals segment to \$10000. These options are generally only used with APW C programs.

-S The -S flag causes a summary to be printed to standard out. This summary shows the total number of segments in the file, the number of each type of OMF record compacted, copied, and created. This information gives you some idea of what changes were made to make the object file smaller.

Compacted object files are smaller and load faster than uncompact load files. The reduction in file size is generally about 40%, although the actual number can vary quite a bit in practice. In addition, if the original file is in OMF 1.0 format, it is converted to OMF 2.0.

Files created with ORCA/M 2.0 are compacted by default. The main reason for using this utility is to convert any old programs you may obtain to the newer OMF format, and to reduce their file size.

COMPILE

```

COMPILE [+D|-D] [+E|-E] [+M|-M] [+L|_L] [+P|-P] [+S|_S] [+T|_T]
        [+W|_W] sourcefile [KEEP=outfile]
        [NAMES=(seg1[ seg2[ ...]])] [language1=(option ...)
        [language2=(option ...) ...]]

```

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

COMPRESS

```

COMPRESS A | C | A C [directory1 [directory2 ...]]

```

This internal command compresses and alphabetizes directories. More than one directory can be specified on a single command line.

- A Use this parameter to alphabetize the file names in a directory. The file names appear in the new sequence whenever you use the CATALOG command.
- C Use this parameter to compress a directory. When you delete a file from a directory, a “hole” is left in the directory that GS/OS fills with the file entry for the next file you create. Use the C parameter to remove these holes from a directory, so that the name of the next file you create is placed at the end of the directory listing instead of in a hole in the middle of the listing.
- A C You can use both the A and C parameters in one command; if you do so, you must separate them with one or more spaces.
- directory* The path name or partial path name of the directory you wish to compress or alphabetize, *not* including any file name. If you do not include a volume or directory path, then the current directory is acted on.

This command works only on GS/OS directories, not on other file systems such as DOS or Pascal. Due to the design of GS/OS, the COMPRESS command will also not work on the disk volume that you boot from – to modify the boot volume of your hard disk, for example, you would have to boot from a floppy disk.

To interchange the positions of two files in a directory, use the SWITCH command.

CONTINUE

```

CONTINUE

```

This command causes control to skip over the remaining statements in the closest nested FOR or LOOP statement. For example, the EXEC file

```
FOR I
  IF {I} == IMPORTANT
    CONTINUE
  END
  DELETE {I}
END
```

would delete all files listed on the command line when the EXEC file is executed except for the file IMPORTANT.

COPY

`COPY [-C] [-F] [-P] [-R] pathname1 [pathname2]`

`COPY [-C] [-F] [-P] [-R] pathname1 [directory2]`

`COPY directory1 directory2`

`COPY [-D] volume1 volume2`

This internal command copies a file to a new subdirectory, or to a duplicate file with a different file name. This command can also be used to copy an entire directory or to perform a block-by-block disk copy.

- C If you specify -C before the first path name, COPY does not prompt you if the target file name (*pathname2*) already exists.
- D If you specify -D before the first path name, both path names are volume names, and both volumes are the same size, then a block-by-block disk copy is performed. Other flags, while accepted, are ignored when this flag is used.
- F Normally, the COPY command copies both the data fork and the resource fork of a file. When the -F flag is used, only the data fork is copied. If the destination file already exists, it's resource fork is left undisturbed. By copying the data fork of a file onto an existing file with a resource fork, it is possible to combine the data fork of the original file with the resource fork of the target file.
- P The COPY command prints progress information showing what file is being copied as it works through a list of files. The -P flag suppresses this progress information.
- R Normally, the COPY command copies both the data fork and the resource fork of a file. When the -R flag is used, only the resource fork is copied. If the destination file already exists, it's data fork is left undisturbed. By copying the resource fork of a file onto an existing file with a data fork, it is possible to add the resource fork of the original file to the data fork of the target file.

pathname1 The full or partial path name (including the file name) of a file to be copied. Wildcard characters may be used in the file name.

- pathname2* The full or partial path name (including the file name) to be given to the copy of the file. Wildcard characters can *not* be used in this file name. If you leave this parameter out, then the current directory is used and the new file has the same name as the file being copied.
- directory1* The path name or partial path name of a directory that you wish to copy. The entire directory (including all the files, subdirectories, and files in the subdirectories) is copied.
- directory2* The path name or partial path name of the directory to which you wish to copy the file or directory. If *directory2* does not exist, it is created (unless *directory1* is empty). If you do not include this parameter, the current directory is used.
- volume1* The name of a volume that you want to copy onto another volume. The entire volume (including all the files, subdirectories, and files in the subdirectories) is copied. If both path names are volume names, both volumes are the same size, *and* you specify the -D parameter, then a block-by-block disk copy is performed. You can use a device name (such as .D1) instead of a volume name.
- volume2* The name of the volume that you want to copy onto. You can use a device name instead of a volume name.

If you do not specify *pathname2*, and a file with the file name specified in *pathname1* exists in the target subdirectory, or if you do specify *pathname2* and a file named *pathname2* exists in the target subdirectory, then you are asked if you want to replace the target file. Type Y and press RETURN to replace the file. Type N and press RETURN to copy the file to the target prefix with a new file name. In the latter case, you are prompted for the new file name. Enter the file name, or press RETURN without entering a file name to cancel the copy operation. If you specify the -C parameter, then the target file is replaced without prompting.

If you do not include any parameters after the COPY command, you are prompted for a path name, since ORCA prompts you for any required parameters. However, since the target prefix and file name are not required parameters, you are *not* prompted for them. Consequently, the current prefix is always used as the target directory in such a case. To copy a file to any subdirectory *other* than the current one, you *must* include the target path name as a parameter either in the command line or following the path name entered in response to the file name prompt.

If you use volume names for both the source and target and specify the -D parameter, then the COPY command copies one volume onto another. In this case, the contents of the target disk are destroyed by the copy operation. The target disk must be initialized (use the INIT command) *before* this command is used. This command performs a block-by-block copy, so it makes an exact duplicate of the disk. Both disks must be the same size and must be formatted using the same FST for this command to work. You can use device names rather than volume names to perform a disk copy. To ensure safe volume copies, it is a good idea to write-protect the source disk.

CREATE

`CREATE directory1 [directory2 ...]`

This internal command creates a new subdirectory. More than one subdirectory can be created with a single command by separating the new directory names with spaces.

directory The path name or partial path name of the subdirectory you wish to create.

CRUNCH

`CRUNCH [-P] rootname`

This external command combines the object files created by partial assemblies or compiles into a single object file. For example, if an assembly and subsequent partial assemblies have produced the object files FILE.ROOT, FILE.A, FILE.B, and FILE.C, then the CRUNCH command combines FILE.A, FILE.B, and FILE.C into a new file called FILE.A, deleting the old object files in the process. The new FILE.A contains only the latest version of each segment in the program. New segments added during partial assemblies are placed at the end of the new FILE.A.

-P Suppresses the copyright and progress information normally printed by the CRUNCH utility.

rootname The full path name or partial path name, including the file name but minus any file name extensions, of the object files you wish to compress. For example, if your object files are named FILE.ROOT, FILE.A, and FILE.B in subdirectory :HARDISK:MYFILES:, you should then use :HARDISK:MYFILES:FILE for *rootname*.

DELETE

`DELETE [-C] [-P] [-W] pathname1 [pathname2 ...]`

This internal command deletes the file you specify. You can delete more than one file with a single command by separating multiple file names with spaces.

-C If you delete the entire contents of a directory by specifying = for the path name, or if you try to delete a directory, the DELETE command asks for confirmation before doing the delete. If you use the -C flag, the delete command does not ask for confirmation before doing the delete.

-P When you delete files using wildcards, or when you delete a directory that contains other files, the delete command lists the files as they are deleted. To suppress this progress information, use the -P flag.

-W When you try to delete a file that does not exist, the DELETE command prints a warning message, but does not flag an error by returning a non-zero status code. If you use the -W flag, the warning message will not be printed.

pathname The full path name or partial path name (including the file name) of the file to be deleted. Wildcard characters may be used in the file name.

If the target file of the DELETE command is a directory, the directory and all of its contents, including any included directories and their contents, are deleted.

DEREZ

```
DEREZ  [-D[EFINE] macro[=data]] [-E[SCAPE]] [-I pathname]
        [-M[AXTRINGSIZE] n] [-O filename]
        [-ONLY typeexpr[(idl[:id2])]] [-P] [-RD]
        [-S[KIP] typeexpr[(idl[:id2])]] [-U[NDEF] macro]
        resourceFile [resourceDescriptionFile]
```

This external command reads the resource fork of an extended file, writing the resources in a text form. This output is detailed enough that it is possible to edit the output, then recompile it with the Rez compiler to create a new, modified resource fork.

`-D[EFINE] macro[=data]` Defines the macro *macro* with the value *data*. This is completely equivalent to placing the statement

```
#define macro data
```

at the start of the first resource description file.

If the optional data field is left off, the macro is defined with a null value.

More than one -d option can be used on the command line.

`-E[SCAPE]` Characters outside of the range of the printing ASCII characters are normally printed as escape sequences, like `\0xC1`. If the -e option is used, these characters are sent to standard out unchanged. Not all output devices have a mechanism defined to print these characters, so using this option may give strange or unusable results.

`-I pathname` Lets you specify one or more path names to search for #include files. This option can be used more than once. If the option is used more than once, the paths are searched in the order listed.

`-M[AXTRINGSIZE] n` This setting controls the width of the output. It must be in the range 2 to 120.

`-O filename` This option provides another way of redirectable the output. It should not be used if command line output redirection is also used. With the -O option, the file is created with a file type of SRC and a language type of Rez.

- ONLY *typeexpr*[(*id1*[:*id2*))]
Lists only resources with a resource type of *typeexpr*, which should be expressed as a numeric value. If the value is followed immediately (no spaces!) by a resource ID number in parenthesis, only that particular resource is listed. To list a range of resources, separate the starting and ending resource ID with a colon.
- P
When this option is used, the copyright, version number, and progress information is written to standard out.
- RD
Suppresses warning messages if a resource type is redeclared.
- S[KIP] *typeexpr*[(*id1*[:*id2*))]
Lists all but the resources with a resource type of *typeexpr*, which should be expressed as a numeric value. If the value is followed immediately (no spaces!) by a resource ID number in parenthesis, only that particular resource is skipped. To skip a range of resources, separate the starting and ending resource ID with a colon.
- U[NDEF] *macro*
This option can be used to undefine a macro variable.

resourceFile This is the name of the extended file to process. The resource fork from this file is converted to text form and written to standard out.

resourceDescriptionFile This file contains a series of declarations in the same format as used by the Rez compiler. More than one resource description file can be used. Any include (not #include), read, data, and resource statements are skipped, and the remaining declarations are used as format specifiers, controlling how DeRez writes information about any particular resource type.

If no resource description file is given, or if DeRez encounters a resource type for which none of the resource description files provide a format, DeRez writes the resource in a hexadecimal format.

The output from DeRez consists of resource and data statements that are acceptable to the Rez resource compiler. If the output from DeRez is used immediately as the input to the resource compiler, the resulting resource fork is identical to the one processed by DeRez. In some cases, the reverse is not true; in particular, DeRez may create a data statement for some input resources.

Numeric values, such as the argument for the -only option, can be listed as a decimal value, a hexadecimal value with a leading \$, as in the ORCA assembler, or a hexadecimal value with a leading 0x, as used by the C language.

For all resource description files specified on the source line, the following search rules are applied:

1. DeRez tries to open the file as is, by appending the file name given to the current default prefix.
2. If rule 1 fails and the file name contains no colons and does not start with a colon (in other words, if the name is truly a file name, and not a path name or partial path name), DeRez appends the file name to each of the path names specified by -i options and tries to open the file.
3. DeRez looks for the file in the folder 13:RInclude.

For more information about resource compiler source files and type declarations, see Chapter 15.

DEVICES

DEVICES [-B] [-D] [-F] [-I] [-L] [-M] [-N] [-S] [-T] [-U] [-V]

The DEVICES command lists all of the devices recognized by GS/OS in a tabular form, showing the device type, device name, and volume name. Various flags can be used to show other information about the devices in an expanded form.

- B Display the block size for block devices.
- D Display the version number of the software driver for the device.
- F Show the number of free blocks remaining on a block device.
- I Display the file system format used by the device.
- L Show all available information about each device. This would be the same as typing all of the other flags.
- M Show the total number of blocks on the device.
- N Display the device number.
- S Display the slot number of the device.
- T Show the type of the device.
- U Show the unit number for the device.
- V Show the volume name for the device.

The name of the device is always displayed, but when you use any flag except -L, the device type and volume name are not shown unless you specifically use the -T and -V flags.

See the GS/OS Technical Reference Manual for a detailed description of what devices are, and what the various fields mean in relation to any particular device.

DISABLE

DISABLE [-P] D | N | B | W | R | I *pathname*

This internal command disables one or more of the access attributes of a GS/OS file.

- P When wildcards are used, a list of the files changed is written to standard out. The -P flag suppresses this progress information.
- D “Delete” privileges. If you disable this attribute, the file cannot be deleted.

N	“Rename” privileges. If you disable this attribute, the file cannot be renamed.
B	“Backup required” flag. If you disable this attribute, the file will not be flagged as having been changed since the last time it was backed up.
W	“Write” privileges. If you disable this attribute, the file cannot be written to.
R	“Read” privileges. If you disable this attribute, the file cannot be read.
I	“Visible” flag. If you disable this attribute, the file will be displayed by the CATALOG command without using the -A flag. In other words, invisible files become visible.
<i>pathname</i>	The full path name or partial path name (including the file name) of the file whose attributes you wish to disable. You can use wildcard characters in the file name.

You can disable more than one attribute at one time by typing the operands with no intervening spaces. For example, to “lock” the file TEST so that it cannot be written to, deleted, or renamed, use the command

```
DISABLE DNW TEST
```

Use the ENABLE command to reenable attributes you disabled with the DISABLE command.

When you use the CATALOG command to list a directory, the attributes that are currently enabled are listed in the access field for each file.

DISKCHECK

```
DISKCHECK volume|device
```

This external command scans the disk for active files and lists all block allocations, including both data and resource forks of any extended file types. It will then notify you of block conflicts, where two or more files are claiming the same block(s), and provide an opportunity to list the blocks and files involved. Finally, it will verify the integrity of the disk's bitmap. Bitmap errors will be reported and you can choose to repair the bitmap.

volume|device The GS/OS volume name or device name of the disk to check. The volume name can be specified with or without a beginning colon or slash; for example,

```
DiskCheck :HardDisk
DiskCheck HardDisk
```

A device name requires a period before the name; for example, .SCSI1. Volume numbers can also be used, as in .D2.

DISKCHECK will only verify a ProDOS volume. It will not work with an HFS volume.

In normal display mode, data scrolls continuously on the screen. While DISKCHECK is running, press the space bar to place DISKCHECK in single step mode. In this mode, block allocations are displayed one at a time, each time the space bar is pressed. Press return to return to normal display mode.

DISKCHECK will check volumes with up to 65535 blocks of 512 bytes (32M).

DISKCHECK makes the following assumptions:

- Blocks zero and one are always used and contain boot code.
- Enough disk integrity exists to make a GetFileInfo call on the volume.
- Block two is the beginning of the volume directory and contains valid information regarding the number of blocks, bitmap locations, entries per block, and entry size.
- All unused bytes at the end of the last bitmap block are truly unused; that is, they will be set to zero whenever the bitmap is repaired.

DISKCHECK may not catch invalid volume header information as an error. Likewise, DISKCHECK does not check all details of the directory structures. Therefore, if large quantities of errors are displayed, it is likely that the volume header information or directory information is at fault.

DUMPOBJ

DUMPOBJ [*option ...*] *pathname* [NAMES=(*seg1 seg2 ...*)]

This external command writes the contents of an object file to standard output (normally the screen). The default format for the listing is object module format (OMF) operation codes and records. You can also list the file as a 65816 machine-language disassembly or as hexadecimal codes.

option You can specify as many of the following options as you wish by separating the options with spaces. If you select two mutually exclusive options (such as +X and +D), the last one listed is used. If an option can't function due to the other options set, it is ignored; for example, if you select -H to suppress segment headers, and also specify -S to select short headers, then the -S is ignored.

- A Suppress all information but the operation codes and operands for each line of an OMF-format or 65816-format disassembly. The default is to include the displacement into the file and the program counter for each line at the beginning of the line.
- +D Write the file dump as a 65816 disassembly rather than OMF records.
- F Suppress the checking of the file type. You can use this option to dump the contents of any file, whether it is in OMF or not. See the following discussion for more information on examining non-OMF files.
- H If the output format is hexadecimal codes (+X option), then this option causes the headers to also be listed as hexadecimal codes. For all other output formats, the headers are not printed at all.

- I For 65816 disassembly listings, assume that the CPU is set to short index (X and Y) registers at the start of the disassembly, rather than starting in full native mode. This option has no effect on OMF-format and hexadecimal listings.
- L Suppress the listing of the body of long constant operands.
- M For 65816 disassembly listings, assume that the CPU is set to short memory (accumulator) registers at the start of the disassembly, rather than starting in full native mode. This option has no effect on OMF-format and hexadecimal listings.
- O Don't show the contents of the segments; list the headers only.
- S Write only the name of the segment and the segment type for the segment headers. The default is to include all of the information in the segment header.
- +X Write the file dump in hexadecimal codes rather than as OMF records. Segment headers are always printed in ASCII text unless you also select the -H option.

pathname The full path name or partial path name (including the file name) of the file you wish to dump. The file may be a library file, the output of an assembler or compiler, a load file, or any other file that conforms to ORCA object module format. If you use the -F option, you can specify a file of any file type.

seg1 seg2 ... The names of specific segments you wish to dump. If you specify the NAMES parameter, only the segments you specify are processed. To get a list of segments in the file, use DUMPOBJ with the -O and -S options. Segment-name searches are case sensitive (that is, *seg1*, *seg2*, ... must match the segment names exactly, including the case of the characters).

If the file consists of more than one segment, each segment is listed separately. Each segment listing begins with the segment header, followed by the segment body. A typical segment header is:


```

Block count      : $00000001      1
Reserved space   : $00000000      0
Length           : $0000001B      27
Kind              : $00          static code
Label length     : $00            0
Number length    : $04            4
Version          : $01            1
Bank size        : $00010000      65536
Org              : $00000000      0
Alignment        : $00000000      0
Number sex       : $00            0
Language card    : $00            0
Segment number   : $0000          0
Segment disp     : $00000000      0
Disp to names    : $002C          44
Disp to body     : $003D          61
Load name        :
Segment name     : SECOND

```

Figure 12.9. Sample DUMPOBJ Segment Header

The format in which the body of the segment is shown depends on the options used. The default is to show the contents of each record in the segment in object module format. A typical OMF segment dump is shown in Figure 12.10. The first column shows the actual displacement into the segment, in bytes, of that record. The segment header takes up 61 bytes (that is, it ends at byte \$3C), so the first record in the segment starts at \$3D. The second column shows the setting of the program counter for that segment; that is, the cumulative number of bytes that the linker will create in the load file. The third and fourth columns show the record type and operation code of the OMF record shown on that line. The last column shows the contents of the record. Expressions are shown in postfix form; that is, the values being acted on are written first, followed by the operator.

The OMF dump is provided to aid in the debugging of compilers; if you are not highly familiar with the OMF, the default DUMPOBJ listing will not be of much use to you. However, you can use the options provided to examine the contents of an object file in machine-language disassembly format or as hexadecimal codes.

00003D 000000	USING (\$E4)	DATA
000043 000000	CONST (\$03)	4BABAE
000047 000003	EXPR (\$EB)	02 : L:MSG2
000050 000005	CONST (\$04)	A00000B9
000055 000009	BEXPR (\$ED)	02 : MSG2
00005E 00000B	CONST (\$04)	DA5A4820
000063 00000F	BEXPR (\$ED)	02 : ~COUT
00006D 000011	CONST (\$0A)	7AFAC8CAD0F1A9000060
000078 00001B	END (\$00)	

Figure 12.10. DUMPOBJ OMF-Format Segment Body

If you select the +D option, the segment body is displayed in 65816 disassembly format. A typical disassembly segment dump is shown in the next table. The first column shows the actual displacement into the segment, in bytes, of the first byte in the line. The second column shows the setting of the program counter for that segment; that is, the cumulative number of bytes that

the linker will create in the load file. The third column shows the disassembly. The disassembly starts with LONGA and LONGI directives indicating whether the disassembler is assuming long or short operands for the accumulator and index registers. ORCA assembly language is described in chapters 17, 18, and 19.

The disassembler tries to keep track of REP and SEP instructions, which are used to set bits in the status register. The status register settings determine whether 16-bit (native mode) or 8-bit (emulation mode) index-register (X and Y) and accumulator-register transfers are used by the CPU. Any time the disassembler finds an REP or SEP instruction, it inserts the appropriate LONGA and LONGI directives in the disassembly to indicate the state of the registers. (The LONGA and LONGI directives tell the ORCA Assembler whether to use long or short operands for transfer instructions.) LONGA and LONGI directives are also placed at the beginning of every segment in the disassembly to indicate the state of the registers on entering the segment. If an expression involving a label was used as the operand of the REP or SEP instruction, then the disassembly might lose track of the setting of the status register.

00003D	000000		LONGA	ON
00003D	000000		LONGI	ON
00003D	000000		SECOND	START
00003D	000000		USING	DATA
000043	000000		PHK	
000045	000001		PLB	
000046	000002		LDX	L:MSG2
00004B	000005		LDY	#\$0000
00004F	000008		LDA	MSG2,Y
000054	00000B		PHX	
000056	00000C		PHY	
000057	00000D		PHA	
000058	00000E		JSR	~COUT
00005D	000011		PLY	
00005F	000012		PLX	
000060	000013		INY	
000061	000014		DEX	
000062	000015		BNE	*+\$F1
000064	000017		LDA	#\$0000
000067	00001A		RTS	
000068	00001B		END	

Figure 12.11. DUMPOBJ Disassembly-Format Segment Body

If you select the +X option, the segment body is displayed in hexadecimal format. A typical hexadecimal segment dump is shown in Figure 12.12. The first column shows the actual displacement into the segment, in bytes, of the first byte in the line. The next four columns show the next 16 bytes in the file. The last column shows the ASCII equivalents of those bytes. The hexadecimal dump starts with the first byte after the segment header (unless you specify the -H option, in which case the segment header is included in the hexadecimal dump), and ends at the last byte before the next segment header. All segments in object files start on block (that is, 512-byte) boundaries, so the bytes from the END record to the end of the block are meaningless (in Figure 12.12 they contain repetitions of the data in the segment).

00003D	E4044441	5441034B	ABAEEB02	84044D53	d DATA K+.k MS
00004D	47320004	A00000B9	ED028304	4D534732	G2 9m MSG2
00005D	0004DA5A	4820ED02	83057E43	4F555400	ZZH m ~COUT
00006D	0A7AFAC8	CAD0F1A9	00006000	434F4E44	zzHJPq) ` COND
00007D	E4044441	5441034B	ABAEEB02	84044D53	d DATA K+.k MS
00008D	47320004	A00000B9	ED028304	4D534732	G2 9m MSG2
00009D	0004DA5A	4820ED02	83057E43	4F555400	ZZH m ~COUT
0000AD	0A7AFAC8	CAD0F1A9	00006000	434F4E44	zzHJPq) ` COND
0000BD	E4044441	5441034B	ABAEEB02	84044D53	d DATA K+.k MS
0000CD	47320004	A00000B9	ED028304	4D534732	G2 9m MSG2
0000DD	0004DA5A	4820ED02	83057E43	4F555400	ZZH m ~COUT
0000ED	0A7AFAC8	CAD0F1A9	00006000	434F4E44	zzHJPq) ` COND
0000FD	E4044441	5441034B	ABAEEB02	84044D53	d DATA K+.k MS
00010D	47320004	A00000B9	ED028304	4D534732	G2 9m MSG2
00011D	0004DA5A	4820ED02	83057E43	4F555400	ZZH m ~COUT
00012D	0A7AFAC8	CAD0F1A9	00006000	434F4E44	zzHJPq) ` COND
00013D	E4044441	5441034B	ABAEEB02	84044D53	d DATA K+.k MS
00014D	47320004	A00000B9	ED028304	4D534732	G2 9m MSG2
00015D	0004DA5A	4820ED02	83057E43	4F555400	ZZH m ~COUT
00016D	0A7AFAC8	CAD0F1A9	00006000	434F4E44	zzHJPq) ` COND
00017D	E4044441	5441034B	ABAEEB02	84044D53	d DATA K+.k MS
00018D	47320004	A00000B9	ED028304	4D534732	G2 9m MSG2
00019D	0004DA5A	4820ED02	83057E43	4F555400	ZZH m ~COUT

Figure 12.12. DUMPOBJ Hexadecimal Format Segment Body

DUMPOBJ can be used to dump the contents of any file, even if it is not in OMF. To dump the contents of a non-OMF file, use the -H and -F options, together with either the +X or +D options.

Any other combination of options, or no options, will probably produce unusable results, since in that case DUMPOBJ attempts to scan the file for segments as if it were in OMF.

DUMPOBJ is extremely useful for debugging compilers and assemblers, and is also useful whenever you want to see the contents of an OMF file.

Some useful aliases you may want to create are:

```

ALIAS DUMP      DUMPOBJ +X -F -H
ALIAS DISASM    DUMPOBJ +D -F -H
ALIAS SCAN      DUMPOBJ -S -O

```

DUMP will dump any file in hexadecimal format. DISASM will assume that the file to be disassembled is in fact 65816 assembly language, and will disassemble it. SCAN will list the segments in an object file.

ECHO

```
ECHO [-N] [-T] string
```

This command lets you write messages to standard output. All characters from the first non-blank character to the end of the line are written to standard out. You can use redirection to write the characters to error out or a disk file.

Shell Reference Manual

-N The **-N** flag suppresses the carriage return normally printed after the string, allowing other output to be written to the same line. One popular use for this option is to write a prompt using the **ECHO** command, then use the **INPUT** command to read a value. With the **-N** flag, the input cursor appears on the same line as the prompt.

-T By default, and tab characters in the string are converted to an appropriate number of spaces before the string is written. If the **-T** flag is used, the tab characters are written as is.

string The characters to write.

If you want to start your string with a space or a quote mark, enclose the string in quote marks. Double the quote marks to imbed a quote in the string. For example,

```
ECHO "   This string starts with 3 spaces and includes a "" character."
```

EDIT

EDIT *pathname1 pathname2 ...*

This external command calls the ORCA editor and opens a file to edit.

pathname1 The full path name or partial path name (including the file name) of the file you wish to edit. If the file named does not exist, a new file with that name is opened. If you use a wildcard character in the file name, the first file matched is opened. If more than one file name is given, up to ten files are opened at the same time.

The ORCA default language changes to match the language of the open file. If you open a new file, that file is assigned the current default language. Use the **CHANGE** command to change the language stamp of an existing file. To change the ORCA default language before opening a new file, type the name of the language you wish to use, and press **RETURN**.

The editor is described in Chapter 13.

ELSE

ELSE

ELSE IF *expression*

This command is used as part of an **IF** command.

ENABLE

ENABLE [-P] D | N | B | W | R | I *pathname*

This internal command enables one or more of the access attributes of a GS/OS file, as described in the discussion of the **DISABLE** command. You can enable more than one attribute at

one time by typing the operands with no intervening spaces. For example, to “unlock” the file TEST so that it can be written to, deleted, or renamed, use the command

ENABLE DNW TEST

When a new file is created, all the access attributes are enabled. Use the ENABLE command to reverse the effects of the DISABLE command. The parameters are the same as those of the DISABLE command.

ENTAB

ENTAB [-L *language*] [*file*]

This external command scans a text stream, converting runs of tabs and space characters into the minimum number of tabs and space characters needed to present the same information on the display screen. Tabs are not used to replace runs of spaces in quoted strings.

-L *language* The ENTAB utility checks the language stamp of the input file and uses the appropriate tab line from the SYSTABS file to determine the location of tab stops. This flag can be used to override the default language number, forcing the utility to use the tab line for some other language. You can use either a language number or a language name as the parameter.

file File to process.

There is no DETAB utility, but the TYPE command can be used to strip tab characters from a file, replacing the tab characters with an appropriate number of space characters.

END

END

This command terminates a FOR, IF, or LOOP command.

ERASE

ERASE [-C] *device* [*name*]

This internal command writes the initialization tracks used by GS/OS to a disk that has already been formatted as a GS/OS disk. In effect, this erases all files on the disk.

-C Normally, the system will ask for permission (check) before erasing a disk. The -C flag disables that check.

device The device name (such as .D1) of the disk drive containing the disk to be formatted; or, if the disk being formatted already has a volume name, you can specify the volume name instead of a device name.

Shell Reference Manual

name The new volume name for the disk. If you do not specify *name*, then the name :BLANK is used.

ORCA recognizes the device type of the disk drive specified by *device*, and uses the appropriate format. ERASE works for all disk formats supported by GS/OS.

ERASE destroys any files on the disk being formatted. The effect of the ERASE command is very similar to the effect of the INIT command, but there are some differences. The INIT command will work on any disk, while the ERASE command can only be used on a disk that has already been initialized. The ERASE command works much faster than the INIT command, since the ERASE command does not need to take the time to create each block on the disk. Finally, when the INIT command is used, each block is filled with zeros. The ERASE command does not write zeros to the existing blocks, so any old information on the disk is not truly destroyed; instead, it is hidden very, very well, just as if all of the files and folders on the disk had been deleted.

EXEC

EXEC

This language command sets the shell default language to the EXEC command language. When you type the name of a file that has the EXEC language stamp, the shell executes each line of the file as a shell command.

EXECUTE

EXECUTE *pathname* [*paramlist*]

This internal command executes an EXEC file. If this command is executed from the ORCA Shell command line, then the variables and aliases defined in the EXEC file are treated as if they were defined on the command line.

pathname The full or partial path name of an EXEC file. This file name cannot include wildcard characters.

paramlist The list of parameters being sent to the EXEC file.

EXISTS

EXISTS *pathname*

This internal command checks to see if a file exists. If the file exists, the {Status} shell variable is set to 1; if the file does not exist, the {Status} shell variable is set to 0. Several disk related errors can occur, so be sure to check specifically for either a 0 or 1 value. When using this command in an EXEC file, keep in mind that a non-zero value for the {Status} variable will cause an EXEC file to abort unless the {Exit} shell variable has been cleared with an UNSET EXIT command.

pathname The full or partial path name of a file. More than one file can be checked at the same time by specifying multiple path names. In this case, the result is zero only if each and every file exists.

EXIT

EXIT [*number*]

This command terminates execution of an EXEC file. If *number* is omitted, the {Status} variable will be set to 0, indicating a successful completion. If *number* is coded, the {Status} variable will be set to the number. This allows returning error numbers or condition codes to other EXEC files that may call the one this statement is included in.

number Exit error code.

EXPORT

EXPORT [*variable1* [*variable2* ...]]

This command makes the specified variable available to EXEC files called by the current EXEC file. When used in the LOGIN file, the variable becomes available at the command level, and in all EXEC files executed from the command level. More than one variable may be exported with a single command by separating the variable names with spaces.

variablen Names of the variables to export.

EXPRESS

EXPRESS [-P] *infile* -O *outfile*

The external command EXPRESS reformats an Apple IIGS load file so that it can be loaded by the ExpressLoad loader that comes with Apple's system disk, starting with version 5.0 of the system disk. When loaded with ExpressLoad, the file will load much faster than it would load using the standard loader; however, files reformatted for use with ExpressLoad can still be loaded by the System Loader.

-P If you specify this option, EXPRESS displays progress information. If you omit it, progress information is not displayed.

infile The full or partial path name of a load file.

-O *outfile* This is the full or partial path name of the file to write. Unlike many commands, this output file is a required parameter.

Since the linker that comes with ORCA can automatically generate a file that is expressed, this utility is generally only used to reformat executable programs you obtain through other sources.

EXPRESS only accepts version 2.0 OMF files as input. You can check the version number of the OMF file using DUMPOBJ, and convert OMF 1.0 files to OMF 2.0 using COMPACT.

ExpressLoad does not support multiple load files; therefore, you cannot use Express with any program that references segments in a run-time library.

The following system loader calls are not supported by ExpressLoad:

- GetLoadSegInfo (\$0F) The internal data structures of ExpressLoad are not the same as those of the System Loader.
- LoadSegNum (\$0B) Because EXPRESS changes the order of the segments in the load file, an application that uses this call and has been converted by EXPRESS cannot be processed by the System Loader. Use the LoadSegName function instead.
- UnloadSegNum (\$0C) Because EXPRESS changes the order of the segments in the load file, an application that uses this call and has been converted by EXPRESS cannot be processed by the System Loader. Use the UnloadSeg (\$0E) function instead.

FILETYPE

`FILETYPE [-P] pathname filetype [auxtype]`

This internal command changes the GS/OS file type, and optionally the auxiliary file type, of a file.

-P When wildcards are used, a list of the files changed is written to standard out. The -P flag suppresses this progress information.

pathname The full path name or partial path name (including the file name) of the file whose file type you wish to change.

filetype The GS/OS file type to which you want to change the file. Use one of the following three formats for *filetype*:

- A decimal number 0-255.
- A hexadecimal number \$00-\$FF.
- The three-letter abbreviation for the file type used in disk directories; for example, S16, OBJ, EXE. A partial list of GS/OS file types is shown in Table 12.13.

auxtype The GS/OS auxiliary file type to which you want to change the file. Use one of the following two formats for *auxtype*:

- A decimal number 0-65535.
- A hexadecimal number \$0000-\$FFFF.

You can change the file type of any file with the FILETYPE command; ORCA does not check to make sure that the format of the file is appropriate. However, the GS/OS call used by the FILETYPE command may disable some of the access attributes of the file. Use the CATALOG command to check the file type and access-attribute settings of the file; use the ENABLE command to reenable any attributes that are disabled by GS/OS.

The linker can automatically set the file type and auxiliary file type of a program; see the descriptions of the {KeepType} and {AuxType} variables in Chapter 14 for details.

<u>Decimal</u>	<u>Hex</u>	<u>Abbreviation</u>	<u>File Type</u>
001	\$01	BAD	Bad blocks file
002	\$02	PCD	Pascal code file (SOS)
003	\$03	PTX	Pascal text file (SOS)
004	\$04	TXT	ASCII text file
005	\$05	PDA	Pascal data file (SOS)
006	\$06	BIN	ProDOS 8 binary load
007	\$07	FNT	Font file (SOS)
008	\$08	FOT	Graphics screen file
009	\$09	BA3	Business BASIC program file (SOS)
010	\$0A	DA3	Business BASIC data file (SOS)
011	\$0B	WPF	Word processor file (SOS)
012	\$0C	SOS	SOS system file (SOS)
015	\$0F	DIR	Directory
016	\$10	RPD	RPS data file (SOS)
017	\$11	RPI	RPS index file (SOS)
176	\$B0	SRC	Source
177	\$B1	OBJ	Object
178	\$B2	LIB	Library
179	\$B3	S16	GS/OS system file
180	\$B4	RTL	Run-time library
181	\$B5	EXE	Shell load file
182	\$B6	STR	load file
184	\$B8	NDA	New desk accessory
185	\$B9	CDA	Classic desk accessory
186	\$BA	TOL	Tool file
200	\$C8	FNT	Font file
226	\$E2	DTS	Defile RAM tool patch
240	\$F0	CMD	ProDOS CI added command file
249	\$F9	P16	ProDOS 16 file
252	\$FC	BAS	BASIC file
253	\$FD	VAR	EDASM file
254	\$FE	REL	REL file
255	\$FF	SYS	ProDOS 8 system load file

Table 12.13. A Partial List of GS/OS File Types

FOR

```
FOR variable [IN value1 value2 ... ]
```

This command, together with the END statement, creates a loop that is executed once for each parameter value listed. Each of the parameters is separated from the others by at least one space. To include spaces in a parameter, enclose it in quote marks. For example, the EXEC file

Shell Reference Manual

```
FOR I IN GORP STUFF "FOO BAR"
    ECHO {I}
END
```

would print

```
GORP
STUFF
FOO BAR
```

to the screen.

If the IN keyword and the strings that follow are omitted, the FOR command loops over the command line inputs, skipping the command itself. For example, the EXEC file named EXECFILE

```
FOR I
    ECHO {I}
END
```

would give the same results as the previous example if you executed it with the command

```
EXECFILE GORP STUFF "FOO BAR"
```

HELP

HELP [*commandname1* [*commandname2* ...]]

This internal command provides on-line help for all the commands in the command table provided with the ORCA development environment. If you omit *commandname*, then a list of all the commands in the command table are listed on the screen.

commandname The name of the ORCA shell command about which you want information.

When you specify *commandname*, the shell looks for a text file with the specified name in the HELP subdirectory in the UTILITIES prefix (prefix 17). If it finds such a file, the shell prints the contents of the file on the screen. Help files contain information about the purpose and use of commands, and show the command syntax in the same format as used in this manual.

If you add commands to the command table, or change the name of a command, you can add, copy, or rename a file in the HELP subdirectory to provide information about the new command.

HISTORY

HISTORY

This command lists the last twenty commands entered in the command line editor. Commands executed in EXEC files are not listed.

HOME

HOME

This command sends a \$0C character to the standard output device. The output can be redirected to files, printers, or error output using standard output redirection techniques.

When the \$0C character is sent to the console output device, the screen is cleared and the cursor is moved to the top left corner of the screen. When the \$0C character is sent to most printers, the printer will skip to the top of the next page.

IF

IF *expression*

This command, together with the ELSE IF, ELSE, and END statements provides conditional branching in EXEC files. The expression is evaluated. If the resulting string is the character 0, the command interpreter skips to the next ELSE IF, ELSE or END statement, and does not execute the commands in between. If the string is anything but the character 0, the statements after the IF statement are executed. In that case, if an ELSE or ELSE IF is encountered, the command skips to the END statement associated with the IF.

The ELSE statement is used to provide an alternate set of statements that will be executed if the main body of the IF is skipped due to an expression that evaluates to 0. It must appear after all ELSE IF statements.

ELSE IF is used to test a series of possibilities. Each ELSE IF clause is followed by an expression. If the expression evaluates to 0, the statements following the ELSE IF are skipped; if the expression evaluates to anything but 0, the statements after the ELSE IF are executed.

As an example, the following code will translate an Arabic digit (contained in the variable {I}) into a Roman numeral.

```

IF {I} == 1
    ECHO I
ELSE IF {I} == 2
    ECHO II
ELSE IF {I} == 3
    ECHO III
ELSE IF {I} == 4
    ECHO IV
ELSE IF {I} == 5
    ECHO V
ELSE
    ECHO The number is too large for this routine.
END

```

INIT

INIT [-C] *device* [*fst*] [*name*]

This external command formats a disk as a GS/OS volume.

Shell Reference Manual

-C	Disable checking. If the disk has been previously initialized, the system will ask for permission (check) before starting initialization. The default is to check.
<i>device</i>	The device name (such as .D1) of the disk drive containing the disk to be formatted; or, if the disk being formatted already has a volume name, you can specify the volume name instead of a device name.
<i>fst</i>	The file system translator number. The default FST is 1 (ProDOS).
<i>name</i>	The new volume name for the disk. If you do not specify <i>name</i> , then the name :BLANK is used.

ORCA recognizes the device type of the disk drive specified by *device*, and uses the appropriate format. INIT works for all disk formats supported by GS/OS.

GS/OS is capable of supporting a wide variety of physical disk formats and operating system file formats. The term file system translator, or FST, has been adopted to refer to the various formats. By default, when you initialize a disk, the INIT command uses the physical format and operating system format that has been in use by the ProDOS and GS/OS operating system since ProDOS was introduced for the Apple //e computer. If you would like to use a different FST, you can specify the FST as a decimal number. Apple has defined a wide variety of numbers for use as FSTs, although there is no reason to expect that all of them will someday be implemented in GS/OS; some of the FST numbers are shown in Table 12.14, and a more complete list can be found in *Apple IIGS GS/OS Reference*, Volume 1. Not all of these FSTs have been implemented in GS/OS as this manual goes to press. Even if an FST has been implemented, not all FSTs can be used on all formats of floppy disks. If you aren't sure if an FST is available, give it a try – if not, you will get an error message.

INIT destroys any files on the disk being formatted.

<u>FST Number</u>	<u>File System</u>
1	ProDOS (Apple II, Apple IIGS) and SOS (Apple ///)
2	DOS 3.3
3	DOS 3.2
4	Apple II Pascal
5	Macintosh MFS
6	Macintosh HFS
7	Lisa
8	Apple CP/M
10	MS/DOS
11	High Sierra
13	AppleShare

Table 12.14 FST Numbers

INPUT

`INPUT variable`

This command reads a line from standard input, placing all of the characters typed, up to but not including the carriage return that marks the end of the line, in the shell variable *variable*.

variable Shell variable in which to place the string read from standard in.

LINK

`LINK [+B|-B] [+C|_C] [+L|_L] [+P|-P] [+S|_S] [+X|-X] objectfile
[KEEP=outfile]`

`LINK [+B|-B] [+C|_C] [+L|_L] [+P|-P] [+S|_S] [+X|-X] objectfile1
objectfile2 ... [KEEP=outfile]`

This internal command calls the ORCA linker to link object files to create a load file. You can use this command to link object files created by assemblers or compilers, and to cause the linker to search library files. The linker is described in Chapter 14.

- +B|-B** The +B flag tells the linker to create a bank relative program. Each load segment in a bank relative program must be aligned to a 64K bank boundary by the loader. When the current version of the Apple IIGS loader loads a bank relative program, it also purges virtually all purgeable memory, which could slow down operations of programs like the ORCA shell, which allows several programs to stay in memory. Bank relative programs take up less disk space than fully relocatable programs, and they load faster, since all two-byte relocation information can be resolved at link time, rather than creating relocation records for each relocatable address.
- +C|-C** Executable files are normally compacted, which means some relocation information is packed into a compressed form. Compacted files load faster and use less room on disk than uncompact files. To create an executable file that is not compacted, use the -C flag.
- +L|-L** If you specify +L, the linker generates a listing (called a link map) of the segments in the object file, including the starting address, the length in bytes (hexadecimal) of each segment, and the segment type. If you specify -L, the link map is not produced.
- +P|-P** The linker normally prints a series of dots as subroutines are processed on pass one and two, followed by the length of the program and the number of executable segments in the program. The -P flag can be used to suppress this progress information.
- +S|-S** If you specify +S, the linker produces an alphabetical listing of all global references in the object file (called a symbol table). If you specify -S, the symbol table is not produced.

+X|-X Executable files are normally expressed, which means they have an added header and some internal fields in the code image are expanded. Expressed files load from disk faster than files that are not expressed, but they require more disk space. You can tell the linker not to express a file by using the **-X** flag.

objectfile The full or partial path name, minus file name extension, of the object files to be linked. All files to be linked must have the same file name (except for extensions), and must be in the same subdirectory. For example, the program TEST might consist of object files named TEST.ROOT, TEST.A, and TEST.B, all located in directory :ORCA:MYPROG:.. In this case, you would use :ORCA:MYPROG:TEST for *objectfile*.

objectfile1 objectfile2,... You can link several object files into one load file with a single LINK command. Enclose in parentheses the full path names or partial path names, minus file name extensions, of all the object files to be included; separate the file names with spaces. Either a .ROOT file or a .A file must be present. For example, the program TEST might consist of object files named TEST1.ROOT, TEST1.A, TEST1.B, TEST2.A, and TEST2.B, all in directory :ORCA:MYPROG:.. In this case, you would use :ORCA:MYPROG:TEST1 for *objectfile1* and :ORCA:MYPROG:TEST2 for *objectfile1*.

You can also use this command to specify one or more library files (GS/OS file type \$B2) to be searched. Any library files specified are searched in the order listed. Only the segments needed to resolve references that haven't already been resolved are extracted from the standard library files.

KEEP=outfile Use this parameter to specify the path name or partial path name of the executable load file.

If you do not use the KEEP parameter, then the link is performed, but the load file is not saved.

If you do not include any parameters after the LINK command, you are prompted for an input file name, as ORCA prompts you for any required parameters. However, since the output path name is not a required parameter, you are *not* prompted for it. Consequently, the link is performed, but the load file is not saved. To save the results of a link, you *must* include the KEEP parameter in the command line or create default names using the {LinkName} variable.

The linker can automatically set the file type and auxiliary file type of the executable file it creates. This is covered in detail in Chapter 14.

To automatically link a program after assembling or compiling it, use one of the following commands instead of the LINK command: ASML, ASMLG, CMPL, CMPLG.

LINKER

LINKER

This language command sets the shell default language for linker script files.

LOOP

LOOP

This command together with the END statement defines a loop that repeats continuously until a BREAK command is encountered. This statement is used primarily in EXEC files. For example, if you have written a program called TIMER that returns a {Status} variable value of 1 when a particular time has been reached, and 65535 for an error, you could cause the program SECURITY.CHECK to be executed each time TIMER returned 1, and exit the EXEC file when TIMER returned 65535. The EXEC file would be

```
UNSET EXIT
LOOP
  TIMER
  SET STAT {STATUS}
  IF {STAT} == 1
    SECURITY.CHECK
  ELSE IF {STAT} == 65535
    BREAK
  END
END
```

MACGEN

```
MACGEN [+C|-C] [-P] infile outfile macrofile1 [macrofile2 ...]
```

This external command (utility) creates a custom macro file for an ORCA assembly-language program by searching one or more macro libraries for the macros referenced in the program and placing the referenced macros in a single file.

- +C|-C If you specify +C (the default value), then all excess spaces and all comments are removed from the macro file to save space. If you use the GEN ON directive (to include expanded macros in your sourcefile listing), or the TRACE ON directive (to include conditional execution directives in your sourcefile listing), then use the -C parameter in MACGEN to improve the readability of the listing.
- P Suppresses the copyright and progress information normally printed by the MACGEN utility.
- infile* The full path name or partial path name (including the file name) of the ORCA assembly-language source file. MACGEN scans *infile* for references to macros.
- outfile* The full path name (including the file name) of the macro file to be created by MACGEN.
- macrofile1 macrofile2 ...* The full path names or partial path names (including the file names) of the macro libraries to be searched for the macros referenced in *infile*. At least one macro library must be specified. Wildcard characters can be used in the file names. If you specify more than one file name, separate the names with one or more spaces.

Since macro-library searches are time consuming, and any given program may use macros from several macro libraries, it is often more efficient to create a custom macro library containing only those macros needed by your program. MACGEN generates such a library.

MACGEN scans *infile*, including all files referenced with COPY and APPEND directives, and builds a list of the macros referenced by the program. If there are still unresolved references to macros, MACGEN scans *macrofile2*, and so on. MACGEN can handle macros that call other macros. If there are still unresolved references to macros after all the macro files you specified in the command line have been scanned, then MACGEN lists the missing macros and prompts you for the name of another macro library. Press RETURN without a file name to terminate the process before all macros have been found. After all macros have been found (or you press RETURN to end the process), *outfile* is created.

The following example scans the file :ORCA:MYPROG for macro names, searches the macro libraries :LIB:MACROS and :LIB:MATHMACS for the referenced macros, and creates the macro file :ORCA:MYMACROS:

```
MACGEN :ORCA:MYPROG :ORCA:MYMACROS :LIB:MACROS :LIB:MATHMACS
```

You can specify a previous version of *outfile* as one of the macro libraries to be searched. For example, suppose the program MYPROG already has a custom macro file called MYMACROS, but you want to add one or more macros from the file LIB.MACROS. In this case, you could use the following command:

```
MACGEN MYPROG MYMACROS MYMACROS LIB.MACROS
```

Before you assemble your program, make sure that the source code contains the directive MCOPY *outfile* to cause the assembler to search *outfile* for the macros.

MAKEBIN

```
MAKEBIN [-P] loadfile [binfile][ORG=val]
```

This external command converts a GS/OS load file to a ProDOS 8 binary load file (file type \$06).

- | | |
|-----------------|--|
| -P | Suppresses the copyright information normally printed by the MAKEBIN utility. |
| <i>loadfile</i> | The full or partial path name of a load file that contains a single static load segment. |
| <i>binfile</i> | The full or partial path name of the binary file you want to create. If you do not specify <i>binfile</i> , <i>loadfile</i> is overwritten with the binary file. |
| ORG= <i>val</i> | The binary file is given a fixed start location at <i>val</i> , and all code is relocated for execution starting at the address <i>val</i> . You can use a decimal number for <i>val</i> , or you can specify a hexadecimal number by preceding <i>val</i> with a dollar sign (\$). If you omit this parameter, <i>loadfile</i> is relocated to start at \$2000. |

The MAKEBIN utility does no checking to make sure that your program will run under ProDOS 8. The load file must consist of a single static load segment. It can be absolute or relocatable. If you include an ORG directive in the source file, that ORG is respected; if there is a source-file ORG and you specify a conflicting ORG in the MAKEBIN command, however, an error occurs and the the binary file is not created.

ORCA does not launch or run binary load files (file type \$06) or ProDOS 8 system load files (file type \$FF). You can use the BLOAD and BRUN commands in AppleSoft BASIC to run these programs.

MAKELIB

```
MAKELIB [-F] [-D] [-P] libfile [ + | - | ^ objectfile1
+ | - | ^ objectfile2 ...]
```

This external command creates a library file.

- F If you specify -F, a list of the file names included in *libfile* is produced. If you leave this option out, no file name list is produced.
- D If you specify -D, the dictionary of symbols in the library is listed. Each symbol listed is a global symbol occurring in the library file. If you leave this option out, no dictionary is produced.
- P Suppresses the copyright and progress information normally printed by the MAKELIB utility.
- libfile* The full path name or partial path name (including the file name) of the library file to be created, read, or modified.
- +*objectfile* The full path name or partial path name (including the file name) of an object file to be added to the library. You can specify as many object files to add as you wish. Separate object file names with spaces.
- objectfile* The file name of a component file to be removed from the library. This parameter is a file name only, not a path name. You can specify as many component files to remove as you wish. Separate file names with spaces.
- ^*objectfile* The full path name or partial path name (including the file name) of a component file to be removed from the library and written out as an object file. If you include a prefix in this path name, the object file is written to that prefix. You can specify as many files to be written out as object files as you wish. Separate file names with spaces.

An ORCA library file (GS/OS file type \$B2) consists of one or more component files, each containing one or more segments. Each library file contains a library-dictionary segment that the linker uses to find the segments it needs.

MAKELIB creates a library file from any number of object files. In addition to indicating where in the library file each segment is located, the library-dictionary segment indicates which object file each segment came from. The MAKELIB utility can use that information to remove

any component files you specify from a library file; it can even recreate the original object file by extracting the segments that made up that file and writing them out as an object file. Use the (-F) and (-D) parameters to list the contents of an existing library file.

The MAKELIB command is for use only with ORCA object-module-format (OMF) library files used by the linker. For information on the creation and use of libraries used by language compilers, consult the manuals that came with those compilers.

MAKELIB accepts either OMF 1 or OMF 2 files as input, but always produces OMF 2 files as output. MAKELIB literally converts OMF 1 files to OMF 2 files before placing them in the library. Among other things, this gives you one way to convert an OMF 1 file to an OMF 2 file: first create a library with the OMF 1 file, then extract the file from the library. The extracted file will be in OMF 2 format.

To create an OMF library file using the ORCA Assembler, use the following procedure:

1. Write one or more source files in which each library subroutine is a separate segment.
2. Assemble the programs, specifying a unique name for each program with the KEEP parameter in the ASSEMBLE command. Each multi-segment program is saved as two object files, one with the extension .ROOT, and one with the extension .A.
3. Run the MAKELIB utility, specifying each object file to be included in the library file. For example, if you assembled two files, creating the object files LIBOBJ1.ROOT, LIBOBJ1.A, LIBOBJ2.ROOT, LIBOBJ2.A, and your library file is named LIBFILE, then your command line should be as follows:

```
MAKELIB LIBFILE +LIBOBJ1.ROOT +LIBOBJ1.A +LIBOBJ2.ROOT +LIBOBJ2.A
```
4. Place the new library file in the LIBRARIES: subdirectory. (You can accomplish this in step 3 by specifying 13:LIBFILE for the library file, or you can use the MOVE command after the file is created.)

MOVE

```
MOVE [-C] [-P] pathname1 [pathname2]
```

```
MOVE [-C] [-P] pathname1 [directory2]
```

This internal command moves a file from one directory to another; it can also be used to rename a file.

-C If you specify -C before the first file name, then MOVE does not prompt you if the target file name (*filename2*) already exists.

-P The MOVE command prints progress information showing what file is being moved as it works through a list of files. The -P flag suppresses this progress information.

pathname1 The full path name or partial path name (including the file name) of the file to be moved. Wildcard characters may be used in this file name.

- pathname2* The full path name or partial path name of the directory you wish to move the file to. If you specify a target file name, the file is renamed when it is moved. Wildcard characters can *not* be used in this path name. If the prefix of *pathname2* is the same as that of *pathname1*, then the file is renamed only.
- directory2* The path name or partial path name of the directory you wish to move the file to. If you do not include a file name in the target path name, then the file is not renamed. Wildcard characters can *not* be used in this path name.

If *pathname1* and the target directory are on the same volume, then ORCA calls GS/OS to move the directory entry (and rename the file, if a target file name is specified). If the source and destination are on different volumes, then the file is copied; if the copy is successful, then the original file is deleted. If the file specified in *pathname2* already exists and you complete the move operation, then the old file named *pathname2* is deleted and replaced by the file that was moved.

NEWER

NEWER *pathname1 pathname2...*

This internal command checks to see if any file in a list of files has been modified since the first file was modified. If the first file is newer than, or as new as, all of the other files, the {Status} shell variable is set to 0. If any of the files after the first file is newer than the first file, the {Status} shell variable is set to 1.

pathname1 The full or partial path name of the file to be checked.

pathname2... The full or partial path name of the files to compare with the first file. If any of the files in this list have a modification date after *pathname1*, {Status} is set to 1.

This command is most commonly used in script files to create sophisticated scripts that automatically decide when one of several files in a project need to be reassembled.

The GS/OS operating system records the modification date to the nearest minute. It is quite possible, unfortunately, to make changes to more than one file, then attempt to rebuild a file, in less than one minute. In this case, the command may miss a file that has been changed. See the TOUCH command for one way to update the time stamp.

Wildcards may be used in any path name. If the first file is specified with a wildcard, only the first matching file is checked. If wildcards are used in the remaining names, each matching file is checked against the first file.

It is possible for the NEWER command to return a value other than 0 or 1; this would happen, for example, if a disk is damaged or if one of the files does not exist at all. For this reason, your script files should check for specific values of 0 or 1.

A status variable other than zero generally causes a script file to exit. To prevent this, be sure and unset the exit shell variable.

PREFIX

PREFIX [-C] [*n*] *directory*[:]

This internal command sets any of the eight standard GS/OS prefixes to a new subdirectory.

- | | |
|------------------|---|
| -C | The PREFIX command does not normally allow you to set a prefix to a path name that does not exist or is not currently available. The -C flag overrides this check, allowing you to set the prefix to any valid GS/OS path name. |
| <i>n</i> | A number from 0 to 31, indicating the prefix to be changed. If this parameter is omitted, 8 is used. This number must be preceded by one or more spaces. |
| <i>directory</i> | The full or partial path name of the subdirectory to be assigned to prefix <i>n</i> . If a prefix number is used for this parameter, you must follow the prefix number with the : character. |

Prefix 8 is the current prefix; all shell commands that accept a path name use prefix 8 as the default prefix if you do not include a colon (:) at the beginning of the path name. Prefixes 9 through 17 are used for specific purposes by ORCA, GS/OS and the Apple IIGS tools; see the section “Standard Prefixes” in this chapter for details. The default settings for the prefixes are shown in Table 12.3. Prefixes 0 to 7 are obsolete ProDOS prefixes, and should no longer be used. Use the SHOW PREFIX command to find out what the prefixes are currently set to.

The prefix assignments are reset to the defaults each time ORCA is booted. To use a custom set of prefix assignments every time you start ORCA, put the PREFIX commands in the LOGIN file.

PRODOS

PRODOS

This language command sets the ORCA shell default language to GS/OS text. GS/OS text files are standard ASCII files with GS/OS file type \$04; these files are recognized by GS/OS as text files. ORCA TEXT files, on the other hand, are standard ASCII files with GS/OS file type \$B0 and an ORCA language type of TEXT. The ORCA language type is not used by GS/OS.

QUIT

QUIT

This internal command terminates the ORCA program and returns control to GS/OS. If you called ORCA from another program, GS/OS returns you to that program; if not, GS/OS prompts you for the next program to load.

RENAME

RENAME *pathname1* *pathname2*

This internal command changes the name of a file. You can also use this command to move a file from one subdirectory to another on the same volume.

pathname1 The full path name or partial path name (including the file name) of the file to be renamed or moved. If you use wildcard characters in the file name, the first file name matched is used.

pathname2 The full path name or partial path name (including the file name) to which *pathname1* is to be changed or moved. You cannot use wildcard characters in the file name.

If you specify a different subdirectory for *pathname2* than for *pathname1*, then the file is moved to the new directory and given the file name specified in *pathname2*.

The subdirectories specified in *pathname1* and *pathname2* must be on the same volume. To rename a file and move it to another volume, use the MOVE command.

RESEQUAL

RESEQUAL [-P] *pathname1* *pathname2*

The external command RESEQUAL compares the resources in two files and writes their differences to standard out.

RESEQUAL checks that each file contains resources of the same type and identifier as the other file; that the size of the resources with the same type and identifier are the same; and that their contents are the same.

-P If this flag is used, a copyright message and progress information is written to error out.

pathname1 The full or partial path name of one of the two files to compare.

pathname2 The full or partial path name of one of the two files to compare.

If a mismatch is found, the mismatch and the subsequent 15 bytes are written to standard out. RESEQUAL then continues the comparison, starting with the byte following the last byte displayed. The following messages appear when reporting differences:

- In 1 but not in 2

The resource type and ID are displayed.

- In 2 but not in 1

The resource type and ID are displayed.

- Resources are different sizes

The resource type, resource ID, and the size of the resource in each file are displayed.

- Resources have different contents

This message is followed by the resource type and ID, then by the offset in the resource, and 16 bytes of the resource, starting at the byte that differed. If more than ten differences are found in the same resource, the rest of the resource is skipped and processing continues with the next resource.

REZ

REZ

This language command sets the default language to Rez. The resource compiler is described in Chapter 15.

RUN

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

This internal command compiles (or assembles), links, and runs a source file. Its function is identical to that of the ASMLG command. See the ASML command for a description of the parameters. See your compiler or assembler manual for the default values of the parameters and the language-specific options available.

SET

```
SET [variable [value]]
```

This command allows you to assign a value to a variable name. You can also use this command to obtain the value of a variable or a list of all defined variables.

variable The variable name you wish to assign a value to. Variable names are not case sensitive, and only the first 255 characters are significant. If you omit *variable*, then a list of all defined names and their values is written to standard output.

value The string that you wish to assign to *variable*. Values are case sensitive and are limited to 65536 characters. All characters, including spaces, starting with the first non-space character after *variable* to the end of the line, are included in *value*. If you include *variable* but omit *value*, then the current value of *variable* is written to standard output. Embed spaces within *value* by enclosing *value* in double quote marks.

A variable defined with the SET command is normally available only in the EXEC file where it is defined, or if defined on the command line, only from the command line. The variable and its value are not normally passed on to EXEC files, nor are the variables set in an EXEC file available to the caller of the EXEC file.

To pass a variable and its value on to an EXEC file, you must export the variable using the EXPORT command. From that time on, any EXEC file will receive a copy of the variable. Note that this is a copy: UNSET commands used to destroy the variable, or SET commands used to change it, will not affect the original. Variables exported from the LOGIN file are exported to the command level.

You can cause changes to variables made in an EXEC file to change local copies. See the EXECUTE command for details.

Use the UNSET command to delete the definition of a variable.

Certain variable names are reserved; see page 96 for a list of reserved variable names.

SHOW

SHOW [LANGUAGE] [LANGUAGES] [PREFIX] [TIME] [UNITS]

This internal command provides information about the system.

LANGUAGE Shows the current system-default language.

LANGUAGES Shows a list of all languages defined in the language table, including their language numbers.

PREFIX Shows the current subdirectories to which the GS/OS prefixes are set. See the section “Standard Prefixes” in this chapter for a discussion of ORCA prefixes.

TIME Shows the current time.

UNITS Shows the available units, including device names and volume names. Only those devices that have formatted GS/OS volumes in them are shown. To see the device names for all of your disk drives, make sure that each drive contains a GS/OS disk.

More than one parameter can be entered on the command line; to do so, separate the parameters by one or more spaces. If you enter no parameters, you are prompted for them.

SHUTDOWN

SHUTDOWN

This internal command shuts down the computer, ejecting floppy disks and leaving any RAM disk intact. A dialog will appear which allows you to restart the computer.

Technically, the command performs internal clean up of the shell's environment, just as the QUIT command does, ejects all disks, and then does an OSShutdown call with the shut down flags set to 0.

SWITCH

SWITCH [-P] *pathname1* *pathname2*

This internal command interchanges two file names in a directory.

-P When wildcards are used, the names of the two files switched are written to standard out. The -P flag suppresses this progress information.

pathname1 The full path name or partial path name (including the file name) of the first file name to be moved. If you use wildcard characters in the file name, the first file name matched is used.

pathname2 The full path name or partial path name (including the file name) to be switched with *pathname1*. The prefix in *pathname2* must be the same as the prefix in *pathname1*. You cannot use wildcard characters in this file name.

For example, suppose the directory listing for :ORCA:MYPROGS: is as follows in the figure below:

:ORCA:MYPROGS:= Name	Type	Blocks	Modified	Created	Access	Subtype
C.SOURCE	SRC	5	26 MAR 86 07:43	29 FEB 86 12:34	DNBWR	C
COMMAND.FILE	SRC	1	9 APR 86 19:22	31 MAR 86 04 22	DNBWR	EXE
ABS.OBJECT	OBJ	8	12 NOV 86 15:02	4 MAR 86 14:17	NBWR	

Figure 12.15. CATALOG :ORCA:MYPROGS: command

To reverse the positions in the directory of the last two files, use the following command:

SWITCH :ORCA:MYPROGS:COMMAND.FILE :ORCA:MYPROGS:ABS.OBJECT

Now if you list the directory again, it looks like this:

:ORCA:MYPROGS:= Name	Type	Blocks	Modified	Created	Access	Subtype
C.SOURCE	SRC	5	26 MAR 86 07:43	29 FEB 86 12:34	DNBWR	C
ABS.OBJECT	OBJ	8	12 NOV 86 15:02	4 MAR 86 14:17	NBWR	
COMMAND.FILE	SRC	1	9 APR 86 19:22	31 MAR 86 04 22	DNBWR	EXE

Figure 12.16. CATALOG :ORCA:MYPROGS: command

You can alphabetize GS/OS directories with the COMPRESS command, and list directories with the CATALOG command. This command works only on GS/OS directories, not on other file systems such as DOS or Pascal. Due to the design of GS/OS, the SWITCH command will also not work on the disk volume that you boot from – to modify the boot volume of your hard disk, for example, you would have to boot from a floppy disk.

TEXT

TEXT

This language command sets the ORCA shell default language to ORCA TEXT. ORCA text files are standard-ASCII files with GS/OS file type \$B0 and an ORCA language type of TEXT. The TEXT file type is provided to support any text formatting programs that may be added to ORCA. TEXT files are shown in a directory listing as SRC files with a subtype of TEXT.

Use the PRODOS command to set the language type to GS/OS text; that is, standard ASCII files with GS/OS file type \$04. PRODOS text files are shown in a directory listing as TXT files with no subtype.

TOUCH

TOUCH [-P] *pathname*

This internal command "touches" a file, changing the file's modification date and time stamp to the current date and time, just as if the file had been loaded into the editor and saved again. The contents of the file are not affected in any way.

-P When wildcards are used, a list of the files touched is written to standard out. The -P flag suppresses this progress information.

pathname The full path name or partial path name (including the file name) of the file to be touched. You can use wildcard characters in this file name, in which case every matching file is touched. You can specify more than one path name in the command; separate path names with spaces.

TYPE

TYPE [+N|-N] [+T|-T] *pathname1* [*startline1* [*endline1*]] [*pathname2* [*startline2* [*endline2*]]...]]

This internal command prints one or more text or source files to standard output (usually the screen).

+N|-N If you specify +N, the shell precedes each line with a line number. The default is -N: no line numbers are printed.

+T|-T The TYPE command normally expands tabs as a file is printed; using the -T flag causes the TYPE command to send tab characters to the output device unchanged.

pathname The full path name or partial path name (including the file name) of the file to be printed. You can use wildcard characters in this file name, in which case every text or source file matching the wildcard file name specification is printed. You can specify more than one path name in the command; separate path names with spaces.

Shell Reference Manual

start linen The line number of the first line of this file to be printed. If this parameter is omitted, then the entire file is printed.

endlinen The line number of the last line of this file to be printed. If this parameter is omitted, then the file is printed from *startline* to the end of the file.

ORCA text files, GS/OS text files, and ORCA source files can be printed with the TYPE command. Use the TYPE command and output redirection to merge files. For example, to merge the files FILE1 and FILE2 into the new file FILE3, use the command:

```
TYPE FILE1 FILE2 > FILE3
```

Normally, the TYPE command functions as a DETAB utility, expanding tabs to an appropriate number of spaces as the file is sent to the output device. The TYPE command examines the language stamp of the file being typed, reading the appropriate tab line from the SYSTABS file to determine where the tab stops are located.

If you are using the type command to append one file to the end of another, you may not want tabs to be expanded. In that case, the -T flag can be used to suppress tab expansions.

UNALIAS

```
UNALIAS variable1 [variable2 ...]
```

The UNALIAS command deletes an alias created with the ALIAS command. More than one alias can be deleted by listing all of them, separated by spaces.

UNSET

```
UNSET variable1 [variable2...]
```

This command deletes the definition of a variable. More than one variable may be deleted by separating the variable names with spaces.

variable The name of the variable you wish to delete. Variable names are not case sensitive, and only the first 255 characters are significant.

Use the SET command to define a variable.

*

* *string*

The * command is the comment. By making the comment a command that does nothing, you are able to rename it to be anything you wish. Since it is a command, the comment character must be followed by a space. All characters from there to the end of the line, or up to a ; character, which indicates the start of the next command, are ignored.

Chapter 13

The Text Editor

The ORCA editor allows you to write and edit source and text files. A brief introduction to the use of the editor was given in Chapter 3. This chapter provides reference material on the editor, including detailed descriptions of all editing commands.

The first section in this chapter, "Modes," describes the different modes in which the editor can operate. The second section, "Macros," describes how to create and use editor macros, which allow you to execute a string of editor commands with a single keystroke. The third section, "Using Editor Dialogs," gives a general overview of how the mouse and keyboard are used to manipulate dialogs. The next section, "Commands," describes each editor command and gives the key or key combination assigned to the command. The last section, "Setting Editor Defaults," describes how to set the defaults for editor modes and tab settings for each language.

Modes

The behavior of the ORCA editor depends on the settings of several modes, as follows:

- Insert.
- Escape.
- Auto Indent.
- Text Selection.
- Hidden Characters.

Most of these modes has two possible states; you can toggle between the states while in the editor. The default for these modes can be changed by changing flags in the SYSTABS file; this is described later in this chapter, in the section "Setting Editor Defaults." All of these modes are described in this section.

Insert

When you first start the editor, it is in over strike mode; in this mode the characters you type replace any characters the cursor is on. In insert mode, any characters you type are inserted at the left of the cursor; the character the cursor is on and any characters to the right of the cursor are moved to the right.

The maximum number of characters the ORCA editor will display on a single line is 255 characters, and this length can be reduced by appropriate settings in the tab line. If you insert enough characters to create a line longer than 255 characters, the line is wrapped and displayed as more than one line. Keep in mind that most languages limit the number of characters on a single source line to 255 characters, and may ignore any extra characters or treat them as if they were on a new line.

To enter or leave the insert mode, type `OE`. When you are in insert mode, the cursor will be an underscore character that alternates with the character in the file. In over strike mode, the cursor is a blinking box that changes the underlying character between an inverse character (black on white) and a normal character (white on black).

Escape

When you press the `ESC` key, the editor enters the escape mode. For the most part, the escape mode works like the normal edit mode. The principle difference is that the number keys allows you to enter repeat counts, rather than entering numbers into the file. After entering a repeat count, a command will execute that number of times.

For example, the `OB` command inserts a blank line in the file. If you would like to enter fifty blank lines, you would enter the escape mode, type `50OB`, and leave the escape mode by typing the `ESC` key a second time.

Earlier, it was mentioned that the number keys were used in escape mode to enter repeat counts. In the normal editor mode, `O` followed by a number key moves to various places in the file. In escape mode, the `O` key modifier allows you to type numbers.

The only other difference between the two modes is the way `CTRL_` works. This key is used primarily in macros. If you are in the editor mode, `CTRL_` places you in escape mode. If you are in escape mode, it does nothing. In edit mode, `OCTRL_` does nothing; in escape mode, it returns you to edit mode. This lets you quickly get into the mode you need to be in at the start of an editor macro, regardless of the mode you are in when the macro is executed.

The remainder of this chapter describes the standard edit mode.

Auto Indent

You can set the editor so that `RETURN` moves the cursor to the first column of the next line, or so that it follows indentations already set in the text. If the editor is set to put the cursor on column 1 when you press `RETURN`, then changing this mode causes the editor to put the cursor on the first non-space character in the next line; if the line is blank, then the cursor is placed under the first non-space character in the first non-blank line above the cursor. The first mode is generally best for line-oriented languages, like assembly language or BASIC. The second is handy for block-structured languages like C or Pascal.

To change the return mode, type `ORETURN`.

Select Text

You can use the mouse or the keyboard to select text in the ORCA editor. This section deals with the keyboard selection mechanism; see "Using the Mouse," later in this chapter, for information about selecting text with the mouse.

The Cut, Copy, Delete and Block Shift commands require that you first select a block of text. The ORCA editor has two modes for selecting text: line-oriented and character-oriented selects. As you move the cursor in line-oriented select mode, text or code is marked a line at a time. In the character-oriented select mode, you can start and end the marked block at any character. Line-oriented select mode is the default for assembly language; for text files and most high-level languages, character-oriented select mode is the default.


While in either select mode, the following cursor-movement commands are active:

- bottom of screen
- top of screen
- cursor down
- cursor up
- start of line
- screen moves

In addition, while in character-oriented select mode, the following cursor-movement commands are active:

- cursor left
- cursor right
- end of line
- tab
- tab left
- word right
- word left

As you move the cursor, the text between the original cursor position and the final cursor position is marked (in inverse characters). Press `RETURN` to complete the selection of text. Press `ESC` to abort the operation, leave select mode, and return to normal editing.

To switch between character- and line-oriented selection while in the editor, type `CTRL-x`.

Hidden Characters

There are cases where line wrapping or tab fields may be confusing. Is there really a new line, or was the line wrapped? Do those eight blanks represent eight spaces, a tab, or some combination of spaces and tabs? To answer these questions, the editor has an alternate display mode that shows hidden characters. To enter this mode, type `C=`; you leave the mode the same way. While you are in the hidden character mode, end of line characters are displayed as the mouse text return character. Tabs are displayed as a right arrow where the tab character is located, followed by spaces until the next tab stop.

Macros

You can define up to 26 macros for the ORCA editor, one for each letter on the keyboard. A macro allows you to substitute a single keystroke for up to 128 predefined keystrokes. A macro can contain both editor commands and text, and can call other macros.

To create a macro, press `⌘ESC`. The current macro definitions for A to J appear on the screen. The `LEFT-ARROW` and `RIGHT-ARROW` keys can be used to switch between the three pages of macro definitions. To replace a definition, press the key that corresponds to that macro, then type in the new macro definition. You must be able to see a macro to replace it - use the left and right arrow keys to get the correct page. Press `OPTION ESC` to terminate the macro definition. You can include `CTRLkey` combinations, `⌘key` combinations, `OPTIONkey` combinations, and the `RETURN`, `ENTER`, `ESC`, and arrow keys. The following conventions are used to display keystrokes in macros:

<code>CTRLkey</code>	The uppercase character <i>key</i> is shown in inverse.
<code>⌘key</code>	An inverse A followed by <i>key</i> (for example, <code>⌘K</code>)
<code>OPTIONkey</code>	An inverse B followed by <i>key</i> (for example, <code>⌥K</code>)
<code>ESC</code>	An inverse left bracket (<code>CTRL [</code>).
<code>RETURN</code>	An inverse M (<code>CTRL M</code>).
<code>ENTER</code>	An inverse J (<code>CTRL J</code>).
<code>UP-ARROW</code>	An inverse K (<code>CTRL K</code>).
<code>DOWN-ARROW</code>	An inverse J (<code>CTRL J</code>).
<code>LEFT-ARROW</code>	An inverse H (<code>CTRL H</code>).
<code>RIGHT-ARROW</code>	An inverse U (<code>CTRL U</code>).
<code>DELETE</code>	A block

Each `⌘key` combination or `OPTIONkey` combination counts as two keystrokes in a macro definition. Although an `⌘key` combination looks (in the macro definition) like a `CTRL A` followed by *key*, and an `OPTIONkey` combination looks like a `CTRL B` followed by *key*, you cannot enter `CTRL A` when you want an `⌘` or `CTRL B` when you want an `OPTION` key.

If you make a mistake typing a macro definition, you can back up with `⌘DELETE`. If you wish to retype the macro definition, press `OPTION ESC` to terminate the definition, press the letter key for the macro you want to define, and begin over. When you are finished entering macros, press `OPTION ESC` to terminate the last option definition, then press `OPTION` to end macro entry. If you have entered any new macro definitions, a dialog will appear asking if you want to save the macros to disk; select `OK` to save the new macro definitions, and `Cancel` to return

to the editor. If you select Cancel, the macros you have entered will remain in effect until you leave the editor.

Macros are saved on disk in the file SYSEMAC in the ORCA shell prefix (prefix 15; see the section on prefixes in Chapter 12).

To execute a macro, hold down **OPTION** and press the key corresponding to that macro.

Using Editor Dialogs

The text editor makes use of a number of dialogs for operations like entering search strings, selecting a file to open, and informing you of error conditions. The way you select options, enter text, and execute commands in these dialogs is the same for all of them.

Figure 13.1 shows the Search and Replace dialog, one of the most comprehensive of all of the editor's dialogs, and one that happens to illustrate many of the controls used in dialogs.

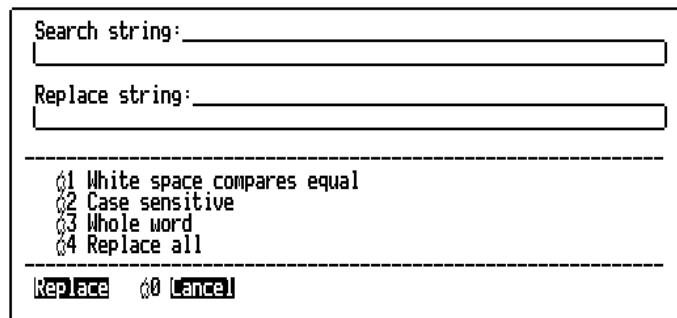


Figure 13.1

The first item in this dialog is an editline control that lets you enter a string. When the dialog first appears, the cursor is at the beginning of this line. You can use any of the line editing commands from throughout the ORCA programming environment to enter and edit a string in this editline control; these line editing commands are summarized in Table 13.2.

<u>command</u>	<u>command name and effect</u>
LEFT-ARROW	cursor left - The cursor will move to the left.
RIGHT-ARROW	cursor right - The cursor will move to the right.
⌘> or ⌘.	end of line - The cursor will move to the right-hand end of the string.
⌘< or ⌘,	start of line - The cursor will move to the left-hand end of the string.
⌘Y or CTRL Y	delete to end of line - Deletes characters from the cursor to the the end of the line.
⌘Z or CTRL Z	undo - Resets the string to the starting string.

ESC or CTRLX	exit - Stops string entry, leaving the dialog without changing the default string or executing the command.
CE or CTRL E	toggle insert mode - Switches between insert and over strike mode. The dialog starts out in the same mode as the editor, but switching the mode in the dialog does not change the mode in the editor.
DELETE	delete character left - Deletes the character to the left of the cursor, moving the cursor left.

Table 13.2 Editline Control Commands

The Search and Replace dialog has two editline items; you can move between them using the tab key. You may also need to enter a tab character in a string, either to search specifically for a string that contains an imbedded tab character, or to place a tab character in a string that will replace the string once it is found. To enter a tab character in an editline string, use **CTab**. While only one space will appear in the editline control, this space does represent a tab character.

Four options appear below the editline controls. Each of these options is preceded by an **C** character and a number. Pressing **Cx**, where x is the number, selects the option, and causes a check mark to appear to the left of the option. Repeating the operation deselects the option, removing the check mark. You can also select and deselect options by using the mouse to position the cursor over the item, anywhere on the line from the **C** character to the last character in the label.

At the bottom of the dialog is a pair of buttons; some dialogs have more than two, while some have only one. These buttons cause some action to occur. In general, all but one of these buttons will have an **C** character and a number to the left of the button. You can select a button in one of several ways: by clicking on the button with the mouse, by pressing the RETURN key (for the default button, which is the one without an **C** character), by pressing **Cx**, or by pressing the first letter of the label on the button. (For dialogs with an editline item, the last option is not available.)

Once an action is selected by pressing a button, the dialog will vanish and the action will be carried out.

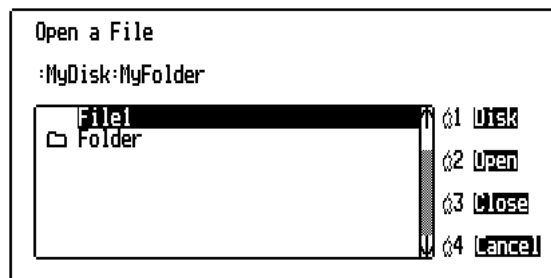


Figure 13.3

Figure 13.3 shown the Open dialog. This dialog contains a list control, used to display a list of files and folders.

You can scroll through the list by clicking on the arrows with the mouse, dragging the thumb with the mouse (the thumb is the space in the gray area between the up and down arrows), clicking in the gray area above or below the thumb, or by using the up and down arrow keys.

If there are any files in the list, one will always be selected. For commands line Open that require a file name, you will be able to select any file in the list; for commands like New, that present the file list so you know what file names are already in use, only folders can be selected. You can change which file is selected by clicking on another file or by using the up or down arrow keys. If you click on the selected name while a folder is selected, the folder is opened. If you click on a selected file name, the file is opened.

Using the Mouse

All of the features of the editor can be used without a mouse, but the mouse can also be used for a number of functions. If you prefer not to use a mouse, simply ignore it. You can even disconnect the mouse, and the ORCA editor will perform perfectly as a text-based editor.

The most common use for the mouse is moving the cursor and selecting text. To position the cursor anywhere on the screen, move the mouse. As soon as the mouse is moved, an arrow will appear on the screen; position this arrow where you would like to position the cursor and click.

Several editor commands require you to select some text. With any of these commands, you can select the text before using the command by clicking to start a selection, then dragging the mouse while holding down the button while you move to the other end of the selection. Unlike keyboard selection, mouse selections are always done in character select mode. You can also select words by double-clicking to start the selection, or lines by triple clicking to start the selection. Finally, if you drag the mouse off of the screen while selecting text, the editor will start to scroll one line at a time.

The mouse can also be used to select dialog buttons, change dialog options, and scroll list items in a dialog. See "Using Editor Dialogs" in this chapter for details.

Command Descriptions

This section describes the functions that can be performed with editor commands. The key assignments for each command are shown with the command description.

Screen-movement descriptions in this manual are based on the direction the display screen moves through the file, not the direction the lines appear to move on the screen. For example, if a command description says that the screen scrolls down one line, it means that the lines on the screen move *up* one line, and the next line in the file becomes the bottom line on the screen.

CTRL@

About

Shows the current version number and copyright for the editor. Press any key or click on the mouse to get rid of the About dialog.

CTRLG

Beep the Speaker

The ASCII control character BEL (\$07) is sent to the output device. Normally, this causes the speaker to beep.

↩, or ↩<

Beginning of Line

The cursor is placed in column one of the current line.

↩DOWN-ARROW

Bottom of Screen / Page Down

The cursor moves to the last visible line on the screen, preserving the cursor's horizontal position. If the cursor is already at the bottom of the screen, the screen scrolls down twenty-two lines.

CTRLC or ↩C

Copy

When you execute the Copy command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is written to the file SYSTEMP in the work prefix. (To cancel the Copy operation without writing the block to SYSTEMP, press ESC instead of RETURN.) Use the Paste command to place the copied material at another position in the file.

CTRLW or ↩W

Close

Closes the active file. If the file has been changed since the last update, a dialog will appear, giving you a chance to abort the close, save the changes, or close the file without saving the changes. If the active file is the only open file, the editor exits after closing the file; if there are other files, the editor selects the next file to become the active file.

DOWN-ARROW

Cursor Down

The cursor is moved down one line, preserving its horizontal position. If it is on the last line of the screen, the screen scrolls down one line.

LEFT-ARROW

Cursor Left

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

RIGHT-ARROW

Cursor Right

The cursor is moved right one column. If it is on the end-of-line marker (usually column 80), the command is ignored.

UP-ARROW

Cursor Up

The cursor is moved up one line, preserving its horizontal position. If it is on the first line of the screen, the screen scrolls up one line. If the cursor is on the first line of the file, the command is ignored.

CTRLX or ⌘X**Cut**

When you execute the Cut command, the editor enters select mode, as discussed in the section “Select Text” in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is written to the file SYSTEMP in the work prefix, and deleted from the file. (To cancel the Cut operation without cutting the block from the file, press ESC instead of RETURN). Use the Paste command to place the cut text at another location in the file.

⌘ESC**Define Macros**

The editor enters the macro definition mode. Press OPTION ESC to terminate a definition, and OPTION to terminate macro definition mode. The macro definition process is described in the section “Macros” in this chapter.

⌘DELETE**Delete**

When you execute the delete command, the editor enters select mode, as discussed in the section “Select Text” in this chapter. Use any of the cursor movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is deleted from the file. (To cancel the delete operation without deleting the block from the file, press ESC instead of RETURN.)

CTRLF or ⌘F**Delete Character**

The character that the cursor is on is deleted and put in the Undo buffer (see the description of the Undo command). Characters to the right of the cursor are moved one space to the left to fill in the gap. The last column on the line is replaced by a space.

DELETE or CTRLD**Delete Character Left**

The character to the left of the cursor is deleted, and the character that the cursor is on, as well as the rest of the line to the right of the cursor, are moved 1 space to the left to fill in the gap. If the cursor is in column one and the over strike mode is active, no action is taken. If the cursor is in column one and the insert mode is active, then the line the cursor is on is appended to the line above and the cursor remains on the character it was on before the delete. Deleted characters are put in the undo buffer.

⌘T or CTRLT**Delete Line**

The line that the cursor is on is deleted, and the following lines are moved up one line to fill in the space. The deleted line is put in the Undo buffer (see the description of the Undo command).

CTRLY or ⌘Y**Delete to EOL**

The character that the cursor is on, and all those to the right of the cursor to the end of the line, are deleted and put in the Undo buffer (see the description of the Undo command).

␣G

Delete Word

When you execute the delete word command, the cursor is moved to the beginning of the word it is on, then delete character commands are executed for as long as the cursor is on a non-space character, then for as long as the cursor is on a space. This command thus deletes the word plus all spaces up to the beginning of the next word. If the cursor is on a space, that space and all following spaces are deleted, up to the start of the next word. All deleted characters, including spaces, are put in the Undo buffer (see the description of the Undo command).

␣. or ␣>

End of Line

If the last column on the line is not blank, the cursor moves to the last column. If the last column is blank, then the cursor moves to the right of the last non-space character in the line. If the entire line is blank, the cursor is placed in column 1.

␣? or ␣/

Help

Displays the help file, which contains a short summary of editor commands. Use **ESC** to return to the file being edited.

The help file is a text file called SYSHELP, found in the shell prefix. Since it is a text file, you can modify it as desired.

␣B or CTRLB

Insert Line

A blank line is inserted at the cursor position, and the line the cursor was on and the lines below it are scrolled down to make room. The cursor remains in the same horizontal position on the screen.

␣SPACEBAR

Insert Space

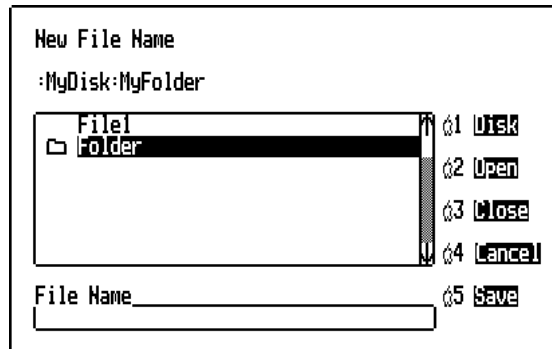
A space is inserted at the cursor position. Characters from the cursor to the end of the line are moved right to make room. Any character in column 255 on the line is lost. The cursor remains in the same position on the screen. Note that the Insert Space command can extend a line past the end-of-line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

CTRLN or ␣N

New

A dialog like the one shown below appears. You need to enter a name for the new file. After entering a name, the editor will open an empty file using one of the ten available file buffers. The file's location on disk will be determined by the directory showing in the dialog's list box.

While the New command requires selecting a file name, no file is actually created until you save the file with the Save command.



CTRL O or ⌘ O

Open

The editor can edit up to ten files at one time. When the open command is used, the editor moves to the first available file buffer, then brings up the dialog shown in Figure 13.4. If there are no empty file buffers, the editor beeps, and the command is aborted.

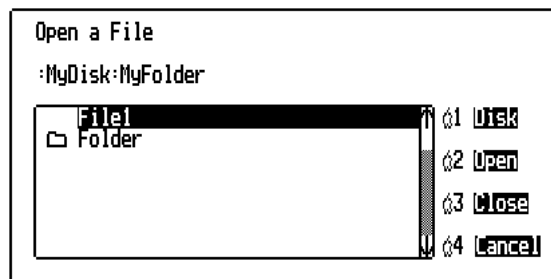


Figure 13.4

Selecting Disk brings up a second dialog that shows a list of the disks available. Selecting one changes the list of files to a list of the files on the selected disk.

When you use the open button, if the selected file in the file list is a TXT or SRC file, the file is opened. If a folder is selected, the folder is opened, and the file list changes to show the files inside the folder. You can also open a file by first selecting a file, then clicking on it with the mouse.

If a folder is open, the close button closes the folder, showing the list of files that contains the folder. You can also close a folder by clicking on the path name shown above the file list. If the file list was created from the root volume of a disk, the close button does nothing.

The cancel button leaves the open dialog without opening a file.

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

CTRLV or ⌘V

Paste

The contents of the SYSTEMP file are copied to the current cursor position. If the editor is in line-oriented select mode, the line the cursor is on and all subsequent lines are moved down to make room for the new material. If the editor is in character-oriented select mode, the material is copied at the cursor column. If enough characters are inserted to make the line longer than 255 characters, the excess characters are lost.

CTRLQ or ⌘Q

Quit

The quit command leaves the editor. If any file has been changed since the last time it was saved to disk, each of the files, in turn, will be made the active file, and the following dialog will appear:

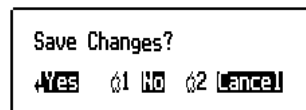


Figure 13.5

If you select Yes, the file is saved just as if the Save command had been used. If you select No, the file is closed without saving any changes that have been made. Selecting Cancel leaves you in the editor with the active file still open, but if several files had been opened, some of them may have been closed before the Cancel operation took effect.

CTRLR or ⌘R

Remove Blanks

If the cursor is on a blank line, that line and all subsequent blank lines up to the next non-blank line are removed. If the cursor is not on a blank line, the command is ignored.

1 to 32767

Repeat Count

When in escape mode, you can enter a *repeat count* (any number from 1 to 32767) immediately before a command, and the command is repeated as many times as you specify (or as many times as is possible, whichever comes first). Escape mode is described in the section “Modes” in this chapter.

RETURN

Return

The RETURN key works in one of two ways, depending on the setting of the auto-indent mode toggle: 1) to move the cursor to column one of the next line; or 2) to place the cursor on the first non-space character in the next line, or, if the line is blank, beneath the first non-space character in the first non-blank line on the screen above the cursor. If the cursor is on the last line on the screen, the screen scrolls down one line.

If the editor is in insert mode, the RETURN key will also split the line at the cursor position.

CTRLA or ⌘A**Save As**

The Save As command lets you change the name of the active file, saving it to a new file name or to the same name in a new file folder. When you use this command, this dialog will appear:

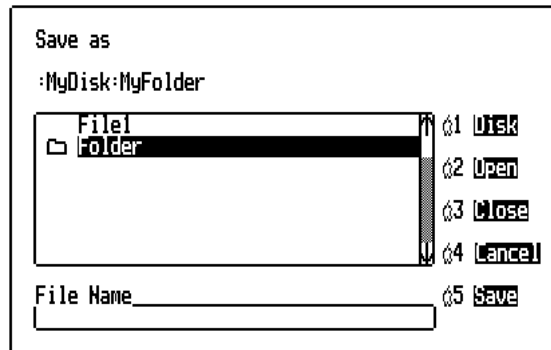


Figure 13.6

Selecting Disk brings up a second dialog that shows a list of the disks available. Selecting one changes the list of files to a list of the files on the selected disk.

When you use the Open button, the selected folder is opened. While using this command, you cannot select any files from the list; only folders can be selected.

If a folder is open, the close button closes the folder, showing the list of files that contains the folder. You can also close a folder by clicking on the path name shown above the file list. If the file list was created from the root volume of a disk, the close button does nothing.

The cancel button leaves the open dialog without opening a file.

The Save button saves the file, using the file name shown in the editline item labeled "File Name." You can also save the file by pressing the RETURN key.

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

CTRLS or ⌘S**Save**

The active file (the one you can see) is saved to disk.

⌘-1 to ⌘-9**Screen Moves**

The file is divided by the editor into 8 approximately equal sections. The screen-move commands move the file to a boundary between one of these sections. The command ⌘1 jumps to the first character in the file, and ⌘9 jumps to the last character in the file. The other seven ⌘*n* commands cause screen jumps to evenly spaced intermediate points in the file.

␣}

Scroll Down One Line

The editor moves down one line in the file, causing all of the lines on the screen to move up one line. The cursor remains in the same position on the screen. Scrolling can continue past the last line in the file.

␣]

Scroll Down One Page

The screen scrolls down twenty-two lines. Scrolling can continue past the last line in the file.

␣{

Scroll Up One Line

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

␣[

Scroll Up One Page

The screen scrolls up twenty-two lines. If the top line on the screen is less than one screen's height from the beginning of the file, the screen scrolls to the beginning of the file.

␣L

Search Down

This command allows you to search through a file for a character or string of characters. When you execute this command, the prompt `Search string:` appears at the bottom of the screen.

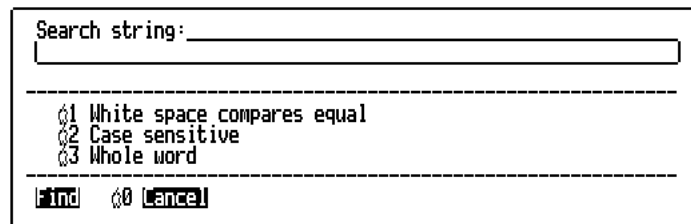


Figure 13.7

If you have previously entered a search string, the previous string appears after the prompt as a default. Type in the string for which you wish to search, and press `RETURN`. The cursor will be moved to the first character of the first occurrence of the search string after the old cursor position. If there are no occurrences of the search string between the old cursor position and the end of the file, an alert will show up stating that the string was not found; pressing any key will get rid of the alert.

By default, string searches are case insensitive, must be an exact match in terms of blanks and tabs, and will match any target string in the file, even if it is a subset of a larger word. All of these defaults can be changed, so we will look at what they mean in terms of how changing the defaults effect the way string searches work.

When you look at a line like


```
lb1   lda   #4
```

without using the hidden characters mode, it is impossible to tell if the spaces between the various fields are caused by a series of space characters, two tabs, or perhaps even a space character or two followed by a tab. This is an important distinction, since searching for `lda<space><space><space>#4` won't find the line if the `lda` and `#4` are actually separated by a tab character, and searching for `lda<tab>#4` won't find the line if the fields are separated by three spaces. If you select the "white space compares equal" option, though, the editor will find any string where `lda` and `#4` are separated by any combination of spaces and tabs, whether you use spaces, tabs, or some combination in the search string you type.

By default, if you search for `lda`, the editor will also find `LDA`, since string searches are case insensitive. In assembly language, that's generally what you want (although not always), but in a language like C, which is case sensitive, you don't usually want to find `LDA` when you type `lda`. Selecting the "case sensitive" option makes the string search case sensitive, so that the capitalization becomes significant. With this option turned on, searching for `lda` would not find `LDA`.

Sometimes when you search for a string, you want to find any occurrence of the string, even if it is imbedded in some larger word. For example, if you are scanning your program for places where it handles spaces, you might enter a string like "space". You would want the editor to find the word `whitespace`, though, and normally it would. If you are trying to scan through a source file looking for all of the places where you used the variable `i`, though, you don't want the editor to stop four times on the word `Mississippi`. In that case, you can select the "whole word" option, and the editor will only stop if it finds the letter `i`, and there is no other letter, number, or underscore character on either side of the letter. These rules match the way languages deal with identifiers, so you can use this option to search for specific variable names – even a short, common one like `i`.

This command searches from the cursor position towards the end of the file. For a similar command that searches back towards the start of the file, see the "Search Up" command.

For a complete description of how to use the mouse or keyboard to set options and move through the dialog, see the section "Using Editor Dialogs" in this chapter.

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

⌘K

Search Up

This command operates exactly like **Search Down**, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file; to search between the current cursor location and the end of the file, use the **Search Down** command.

⌘J

Search and Replace Down

This command allows you to search through a file for a character or string of characters, and to replace the search string with a replacement string. When you execute this command, the following dialog will appear on the screen:

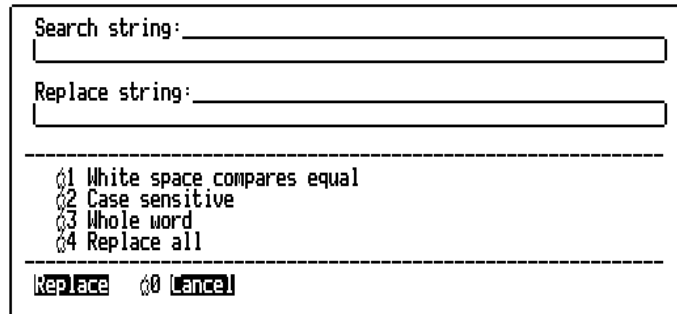


Figure 13.8

The search string, the first three options, and the buttons work just as they do for string searches; for a description of these, see the Search Down command. The replace string is the target string that will replace the search string each time it is found. By default, when you use this command, each time the search string is found in the file you will see this dialog:

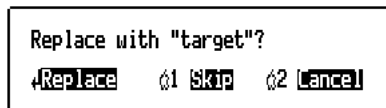


Figure 13.9

If you select the Replace option, the search string is replaced by the replace string, and the editor scans forward for the next occurrence of the search string. Choosing Skip causes the editor to skip ahead to the next occurrence of the search string without replacing the occurrence that is displayed. Cancel stops the search and replace process.

If you use the "replace all" option, the editor starts at the top of the file and replaces each and every occurrence of the search string with the target string. On large files, this can take quite a bit of time. To stop the process, press ⌘. (open-apple period). While the search and replace is going on, you can see a spinner at the bottom right corner of the screen, showing you that the editor is still alive and well.

⌘H

Search and Replace Up

This command operates exactly like Search and Replace Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file; to search between the current cursor location and the end of the file, use the Search and Replace Down command. If you use the "replace all" option, this command works exactly the same way the Search and Replace Down command does when it uses the same option.

⌘-

Select File

The editor can edit up to ten files at one time. When you use this command, a dialog appears showing the names of the ten files in memory. You can then move to one of the files by pressing

⌘n, where n is one of the file numbers. You can exit the dialog without switching files by pressing ESC or RETURN.

See also the Switch Files command.

⌘TAB

Set and Clear Tabs

If there is a tab stop in the same column as the cursor, it is cleared; if there is no tab stop in the cursor column, one is set.

⌘[

Shift Left

If this command is issued when no text is selected, you enter the text selection mode. Pressing RETURN leaves text selection mode.

At any time while text is selected, using the command shifts all of the selected text left one character. This is done by scanning the text, one line at a time, and removing a space right before the first character on each line that is not a space or tab. If the character to be removed is a tab character, it is first replaced by an equivalent number of spaces. If there are no spaces or tabs at the start of the line, the line is skipped.

If a large amount of text is selected, this command may take a lot of time. While the editor is working, you will see a spinner at the bottom right of the screen; this lets you know the editor is still processing text. You can stop the operation by pressing **⌘.**, but this will leave the selected text partially shifted.

⌘]

Shift Right

If this command is issued when no text is selected, you enter the text selection mode. Pressing RETURN leaves text selection mode.

At any time while text is selected, using the command shifts all of the selected text right one character. This is done by scanning the text, one line at a time, and adding a space right before the first character on each line that is not a space or tab. If this leaves the non-space character on a tab stop, the spaces are collected and replaced with a tab character. If a blank line is encountered, no action is taken.

If a large amount of text is selected, this command may take a lot of time. While the editor is working, you will see a spinner at the bottom right of the screen; this lets you know the editor is still processing text. You can stop the operation by pressing **⌘.**, but this will leave the selected text partially shifted.

⌘n

Switch Files

The editor can edit up to ten files at one time. Each of these files is numbered, starting from 0 and proceeding to 9. The numbers are assigned as the files are opened from the command line. To move from one file to the next, press **⌘n**, where n is a numeric key.

When you switch files, the original file is not changed in any way. When you return to the file, the cursor and display will be in the same place, the undo buffer will still be active, and so forth. The only actions that are not particular to a specific file buffer are those involving the clipboard – Cut, Copy and Paste all use the same clipboard, so you can move chunks of text from one file to another.

See also the Select File command.

TAB

Tab

In insert mode, or when in over strike mode and the next tab stop is past the last character in the line, this command inserts a tab character in the source file and moves to the end of the tab field. If you are in the over strike mode and the next tab stop is not past the last character on the line, the Tab command works like a cursor movement command, moving the cursor forward to the next tab stop.

Some languages and utilities do not work well (or at all) with tab stops. If you are using one of these languages, you can tell the editor to insert spaces instead of tab characters; see the section "Setting Editor Defaults," later in this chapter, to find out how this is done.

ÓTAB

Tab Left

The cursor is moved to the previous tab stop, or to the beginning of the line if there are no more tab stops to the left of the cursor. This command does not enter any characters in the file.

ÓRETURN

Toggle Auto Indent Mode

If the editor is set to put the cursor on column one when you press RETURN, it is changed to put the cursor on the first non-space character; if set to the first non-space character, it is changed to put the cursor on column one. Auto-indent mode is described in the section "Modes" in this chapter.

ESC

Toggle Escape Mode

If the editor is in the edit mode, it is put in escape mode; if it is in escape mode, it is put in edit mode. When you are in escape mode, pressing any character not specifically assigned to an escape-mode command returns you to edit mode. Escape and edit modes are described in the section "Modes" in this chapter.

When in escape mode, ÓCTRL_ will return you to edit mode. In edit mode the command has no effect. From edit mode, CTRL_ will place you in escape mode, but the command has no effect in escape mode. These commands are most useful in an editor macro, where you do not know what mode you are in on entry.

CTRL E or ÓE

Toggle Insert Mode

If insert mode is active, the editor is changed to over strike mode. If over strike mode is active, the editor is changed to insert mode. Insert and over strike modes are described in the section "Modes" in this chapter.

CTRL ÓX

Toggle Select Mode

If the editor is set to select text for the Cut, Copy, and Delete commands in units of one line, it is changed to use individual characters instead; if it is set to character-oriented selects, it is toggled to use whole lines. See the section "Modes" in this chapter for more information on select mode.

␣UP-ARROW**Top of Screen / Page Up**

The cursor moves to the first visible line on the screen, preserving the cursor's horizontal position. If the cursor is already at the top of the screen, the screen scrolls up twenty-two lines. If the cursor is at the top of the screen and less than twenty-two lines from the beginning of the file, then the screen scrolls to the beginning of the file.

CTRLZ or ␣Z**Undo Delete**

The last operation that changed the text in the current edit file is reversed, leaving the edit file in the previous state. Saving the file empties the undo buffer, so you cannot undo changes made before the last time the file was saved.

The undo operation acts like a stack, so once the last operation is undone, you can undo the one before that, and so on, right back to the point where the file was loaded or the point where the file was saved the last time.

␣LEFT-ARROW**Word Left**

The cursor is moved to the beginning of the next non-blank sequence of characters to the left of its current position. If there are no more words on the line, the cursor is moved to the last word in the previous line or, if it is blank, to the last word in the first non-blank line preceding the cursor.

␣RIGHT-ARROW**Word Right**

The cursor is moved to the start of the next non-blank sequence of characters to the right of its current position. If there are no more words on the line, the cursor is moved to the first word in the next non-blank line.

Setting Editor Defaults

When you start the ORCA editor, it reads the file named SYSTABS (located in the ORCA shell prefix), which contains the default settings for tab stops, return mode, insert mode, tab mode, and select mode. The SYSTABS file is an ASCII text file that you can edit with the ORCA editor.

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

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

ORCA language numbers are discussed in the section "Command Types/Language Names" in Chapter 12.

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

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

1. If the first column contains a zero, pressing `RETURN` in the editor causes the cursor to go to column one in the next line; if it's a one, pressing `RETURN` sends the cursor to the first non-space character in the next line (or, if the line is blank, beneath the first non-space character in the first non-blank line on the screen above the cursor).
2. If the second character is zero, the editor is set to line-oriented selects; if one, it is set to character-oriented selects.
3. This flag is not used by the current version of the ORCA editor. It should be set to 0.
4. The fourth character is used by the ORCA/Desktop editor, and is used to set the default cursor mode. A zero will cause the editor to start in over strike mode; a one causes the editor to start in insert mode.
5. If the fifth character is a 1, the editor inserts a tab character in the source file when the Tab command is used to tab to a tab stop. If the character is a 0, the editor inserts an appropriate number of spaces, instead.
6. If the sixth character is a 0, the editor will start in over strike mode; if it is a 1, the editor starts in insert mode. Using a separate flag for the text based editor (this one) and the desktop editor (see the fourth flag) lets you enter one mode in the desktop editor, and a different mode in the text based editor.

The third line of each set of lines in the SYSTABS file sets default tab stops. There are 255 zeros and ones, representing the 255 character positions available on the edit line. The ones indicate the positions of the tab stops. A two in any column of this line sets the end of the line; if the characters extend past this marker, the line is wrapped. The column containing the two then replaces the default end-of-line column (the default right margin) when the editor is set to that language.

For example, the following lines define the defaults for ORCA Assembly Language:

[illegible]

The last three lines are actually one long line.

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

- RETURN sends the cursor to column one.

- Line-oriented selects.
- Word wrapping starts in column 80.
- There is a tab stop every eighth column.
- The editor starts in over strike mode.
- Tab characters are inserted to create tabbed text.

Note that you can change tabs and editing modes while in the editor.

Chapter 14

The Link Editor

This chapter describes the use and operation of the link editor. Key points covered in this chapter are:

- How the linker works.
- Link edit command parameters.
- Link editor output.
- Program segmentation.
- Creating library files.
- Using linker script files.

Overview

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 four distinct kinds of files that the ORCA/M system uses:

1. The Source File (SRC). These files contain the source code. They are created using the text editor. Source files are the input to the assembler or compiler.
2. The Object File (OBJ). The assembler or compiler is used to convert the source file into an object file, which in turn is the kind of file the link editor uses. The link editor can relocate the code contained within an object file. Note that the link editor does not know or care what source language produced the object file. A Pascal or BASIC compiler designed for use with this system produces an object file in the same format as that produced by the assembler.
3. The Executable File (usually EXE or S16). Output from the link editor is in the form of an executable file. This file is ready to be executed directly from ORCA; merely enter the file name on the command line. If an ORG directive was not specified during the assembly, then the file produced by the linker is relocatable. The system loader will load a relocatable file into available memory. If the program was ORG'ed to a specific address, then the loader will try to load the file at that address.

4. The Library File (LIB). A special type of object file is a library file. A library is a collection of useful, frequently-used subroutines. These subroutines are kept in a library so that they can be linked with any program that references them. Libraries can be created using the MAKELIB utility, as explained in Chapter 12.

The link editor is invoked by using any shell command that does an assembly or compile, followed by a link edit. These are:

ASML, ASMLG, CMPL, CMPLG, and RUN

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 the object file on disk and loading the file into memory.

As each subroutine is processed by pass one, it is assigned a final position in the executable load file, 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 values are expressed as offsets from the beginning of the load segment. This process is then repeated with the next subroutine.

Once all other object files specified as input have been scanned, the linker checks to see if there are any external references that have not been satisfied. If so, the libraries are scanned. Subroutines can appear in any order in a library. The linker will extract only those segments that are needed. It pulls in segments that have global labels that satisfy the external references it needs. Libraries can themselves reference other libraries, regardless of the order of segments in the library file.

By default, when the linker scans libraries, it scans prefix 13, processing each file with a file type of LIB in turn. If the {Libraries} shell variable has a value, though, the linker instead searches the files listed in the {Libraries} shell variable. For example, after

```
set Libraries :AppleLink:Project:Project.Libs
```

the linker will no longer scan prefix 13 for libraries, but it will scan the library file :AppleLink:Project:Project.Libs.

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. After a load segment has been completed, the linker writes out the relocation dictionary, which tells the loader how to relocate the program when it is time to execute it.

Object Modules Created by the Assembler

When the assembler is directed to save the results of an assembly using the KEEP parameter with a shell command, the {KeepName} variable, or a KEEP assembly directive in the source code, it is necessary to supply a file name. The assembler then creates two files. The first file contains

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 is saved with the file name specified from the command line KEEP parameter or assembler 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 occur in the source file.

If the +M flag was used to assemble the source file, the object modules are written to memory as "unnamed" files. After linking is accomplished, these object modules are discarded. If you will be performing partial compiles, or using separate compilation, do not use the +M flag.

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 reassembled) 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. If a second partial assembly is performed, the reassembled modules are placed in a file with the name OBJECT.C, and so on for additional partial assemblies.

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 OBJ 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 executable 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. Library subroutines may be specified in addition to object files.

If there are still unresolved external references, the link editor assumes that these are to be resolved from library files. The link editor searches the library prefix for library files. If there are no library files to be found, the linker assumes that the unresolved references are errors.

If the library directory contains any library files, each library file is searched once, in the order in which it appears in the catalog. If any subroutine has a name or global label 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. Subroutines may appear in the library in any order.

Having found all of the subroutines it can, the link editor proceeds to pass two of the link edit. Pass two produces an EXE 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 executable 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. Default parameters are indicated by an underline.

```
LINK [+B|-B] [+C|-C] [+L|-L] [+S|-S] [+X|-X] objectfile
      [KEEP=outfile]
```

```
LINK [+B|-B] [+C|-C] [+L|-L] [+S|-S] [+X|-X] objectfile1
      objectfile2 ... [KEEP=outfile]
```

+B|B The +B flag tells the linker to create a bank relative program. Each load segment in a bank relative program must be aligned to a 64K bank boundary by the loader. When the current version of the Apple IIGS loader loads a bank relative program, it also purges virtually all purgeable memory, which could slow down operations of programs like the ORCA shell, which allows several programs to stay in memory. Bank relative programs take up less disk space than fully relocatable programs, and they load faster, since all two-byte relocation information can be resolved at link time, rather than creating relocation records for each relocatable address.

+C|-C Executable files are normally compacted, which means some relocation information is packed into a compressed form. Compacted files load faster and use less room on disk than uncompact files. To create an executable file that is not compacted, use the -C flag.

+L|L If you specify +L, a link map of the segments in the object file (including the starting address, the length in hexadecimal of each segment, and the segment type) is produced.

+S|S If you specify +S, the linker produces an alphabetical listing of all global labels and labels that appeared in data areas.

+X|-X Executable files are normally expressed, which means they have an added header and some internal fields in the code image are expanded. Expressed files load from disk faster than files that are not expressed, but they require more disk space. You can tell the linker not to express a file by using the -X flag.

objectfile The full path name, including file name, minus file name extensions, of the object file to be linked. All modules to be linked must have the same file name, except for extensions, and must be in the same directory. For example, the program TEST might consist of object modules named TEST.ROOT, and TEST.A located in :ORCA:ORSON:. In this case, you would use :ORCA:ORSON:TEST for objectfile. Partial path names, as always, can be used.

objectfile1 objectfile2... You can link several object or library files into one load file with a single LINK command. Include the full or partial path names, minus file name

extensions, of all the object or library files to be included. Separate the path names with blanks. The first file named, *objectfile1*, must have a .ROOT file; for the other object files, the .ROOT file is optional. Note that library files can be specified here, as well as object files.

Keep=outfile Use this parameter to specify the path name or partial path name, including the file name, of the executable load file. If the {LinkName} variable is not used, this parameter is required to produce an executable file.

Specifying the Keep Name with a Shell Variable

The linker will not create an executable program unless you provide a name through some mechanism. One way to give the linker an output file name is by using a keep directive in your assembly language source file, then using one of the commands like RUN that combine the assemble and link step. Another way to specify a link name is by typing the name explicitly when you use the LINK command, RUN command, etc.

Both of these methods, though, require you to provide a file name in every program or on every assemble or link command. By using the shell variables {KeepName} or {LinkName}, you can set up an output name that will be used at all times.

The {KeepName} shell variable provides a default name used by the commands that assemble a program or assemble and link a program. Setting the {KeepName} shell variable is equivalent to typing the same name each and every time the RUN command, CMPL command, and so forth are used. If an explicit keep parameter is coded on one of these commands, though, the value of the {KeepName} shell variable is ignored.

The {LinkName} shell variable is used to provide a different default for executable files. If the {LinkName} shell variable is used, the linker ignores the {KeepName} shell variable, using the value specified by {LinkName} for all executable files, unless a keep parameter is explicitly coded.

There are two special characters used with these variables that affect the automatic naming: % and \$. Using the % will cause the shell to substitute the source file name. Using \$ expands to the file name with the last extension removed (the last period (.) and trailing characters).

Specifying the File Type with a Shell Variable

By default, the linker creates files with a file stamp of EXE and an auxiliary file type of \$0100. You can use the FILETYPE command to change the file type and auxiliary file type after the program has been created, or you can use the {KeepType} shell variable or {AuxType} shell variables to specify the file type and auxiliary file type before creating the program.

The {KeepType} variable should be set to a single value, specified as a hexadecimal or decimal integer, or a three-letter GS/OS file type. The KeepType string sets the file type for the executable file produced by the linker. Legal file types are \$B3 to \$BF. Legal file descriptors are: EXE, S16, RTL, STR, NDA, LDA, TOL, etc.

The {AuxType} shell variable should also be set to a single value, which must be specified as a decimal or hexadecimal integer. The AuxType string sets the auxiliary file type for the executable file produced by the linker. Any value from 0 to 65535 (\$FFFF) can be used. A common setting for the AuxType shell variable is \$DB01, marking the program as GS/OS aware.

Link Editor Output

In addition to generating the load module, the link editor can produce printed output, showing exactly what it did. To get a listing of the segments in the load module, use +L. To get a global symbol table use +S. To explain the format of the linker output, a sample is shown below. The first sample shows the output without the symbol table or segment list. The second shows output with both the global symbol table and segment list. An error was purposely introduced to show how they are listed.

```
Link Editor 2.0

Pass 1: ...
Pass 2: ..
Error at 00000040 past COMMON PC = 0000004A : Unresolved reference Label: CHAR
.

1 error found during link
8 was the highest error level

There is 1 segment, for a length of $000000E2 bytes.
```

Figure 14.1 Link Editor Output without Symbol Table or Segment List

```
Link Editor 2.0 B1

Segment:

00000000 0000000A 01 Code: MAIN
0000000A 00000064 01 Data: COMMON
Error at 00000040 past COMMON PC = 0000004A : Unresolved reference Label: CHAR
0000006E 00000074 01 Code: INIT

Global symbol table:

0000000A G 01 01 COMMON                0000006E G 01 00 INIT
00000000 G 01 00 MAIN

Segment Information:

      Number      Name      Type      Length      Org
      -----
          1              $00      $000000E2  Relocatable

1 error found during link
8 was the highest error level

There is 1 segment, for a length of $000000E2 bytes.
```

Figure 14.2 Link Editor Output with Symbol Table and Segment List

Output With -S and -L Options

During the first and second pass of the linker, there are messages telling you how far the link process has progressed. The dots (.) after the words `Pass 1` and `Pass 2` indicate the number of object module segments that were processed by that pass. In the example shown above, an error occurred during pass two. In this case, the error is an unresolved reference label. This means that the linker could not find a reference to the label `CHAR`. There is a list of error messages and their meanings in Appendix A.

The Segment Table

If `+L` is specified, the code segment table is printed on pass two. Code segments are the segments you create in the assembly language source file by breaking your program up into subroutines using the `START` directive, `DATA` directive, `PRIVATE` directive, and `PRIVDATA` directive. The segment table lists the displacement into the load segment where the program segment starts, the length of the program segment, the load segment number, the segment type, and its name. If an error occurs, the error message appears in the segment table.

At the end of the link, the linker also lists the load segment table. Load segments are the individual pieces of the program loaded into memory by the System Loader or by `ExpressLoad`. Unless you specify a segment name as the operand for the `START` directive, `DATA` directive, `PRIVATE` directive, or `PRIVDATA` directive, the linker only produces one load segment, which has no name and is called the blank segment. This is the load segment you see in Figure 14.2; it has a load segment number of 1, no name, a type of `$00` (a static code segment), a length of `$00E2`, and is relocatable.

The load segment number shown in column 1 of the load segment table can be matched with the load segment numbers from column 3 of the code segment list to find out which code segments ended up in which load segments. In this example, there was only one load segment, so all of these numbers are 1.

Global Symbol Table

If the `+S` parameter was specified on the command line, the listing continues with an alphabetized global symbol table. In the global symbol table, there are three numbers and a letter to the left of each symbol. The first number is hexadecimal; it gives the offset for the symbol. Next comes a letter. It is `G` for symbols that can be used from any source file, and `P` for private symbols, which can only be used from the source file they were defined in. The next number tells which source file a symbol was defined in. The last number indicates whether the symbol is for code or data area (`00` - code, `01`, `02` - data area). If the symbol is in a data area, the number indicates which one. All symbols defined in the same data area have the same number. If errors occurred, the linker then lists the number of errors and the highest error level encountered. The error level tells how severe the error was. Error levels are described in Appendix A. Finally, the linker gives all load segments along with their type, length, and load location.

Program Segmentation

The executable load file produced by the linker consists of one or more load segments. For most programs that are written in assembly language, the entire program is grouped into a single load segment. When the program is executed, the loader (a program built into GS/OS) loads the load segment into a free area of memory, relocating the code as necessary, and executes it. The

program can then make calls to the memory manager to allocate work space from the memory that is still not being used.

Due to the design of the 65816, it is not possible for a program to execute across a bank boundary. (A bank is the 64K of memory, where each byte's address has the same first byte. For example, \$020000, \$020001, ... \$02FFFF are all in bank 2.) Because of this restriction, it is necessary to limit relocatable load segments to 64K. Understanding the technical reasoning is not important here – what is important is to realize that this places an upper limit on the size of a program that consists of a single segment. If your program exceeds 64K in size, you will need to split it into more than one segment. You will know that this is necessary when the linker starts giving the error message `Address is not in current bank`, and the program size is greater than \$FFFF.

To split your program into more than one segment, you must place load segment names after the `START`, `DATA`, `PRIVATE` or `PRIVDATA` directives that start each of your code segments. The syntax for doing this is discussed in the assembler reference manual where these directives are described. The segments do not have to be in any particular order. The linker will create one load segment for each unique load segment name, grouping all of the code segments with the same load segment name into the same load segment. Note that the linker recognizes a special load segment with no name, called the blank segment. It is the blank segment that you normally create when you write a program that does not have load segment names specified in the operand field of the `START`, `DATA`, `PRIVATE` or `PRIVDATA` directives.

As an example, consider the following short program that simply hops around between three segments, called `SEG1`, `SEG2`, and the blank segment.

```

                                mcopy MyProg.Macros
                                keep  Prog
Main    start
        phk
        plb
        js1 Sub1
        js1 Sub2
        js1 Sub3
        lda #0
        rtl
        end

Sub1    start SEG1
        puts #'Hello from Sub1',CR=T
        rtl
        end

Sub2    start SEG2
        puts #'Hello from Sub2',CR=T
        rtl
        end

Sub3    start SEG1
        puts #'Hello from SUB3',CR=T
        rtl
        end

```

The linker will create a program with three load segments. Since the first segment assembled is the blank segment, that is the one that will be executed first. The blank load segment will

consist of the subroutine MAIN. The second segment will be SEG1. It will contain SUB1 and SUB3. The last segment, called SEG2, will contain SUB2.

The loader is free to put segments wherever it finds room in memory. Naturally, that means that you cannot count on the segments being in the same bank of memory, so long addressing must be used. That is why JSL and RTL instructions were used, instead of the shorter and faster JSR and RTS that you would use in a program that consisted of a single segment. Long addressing must also be used on loads and stores that cross into another load segment.

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 using the MAKELIB utility. As an example of the MAKELIB command, suppose that you wish to create an arithmetic library called ARITHLIB containing the four routines add, subtract, multiply, and divide. The process to build ARITHLIB is as follows:

1. Create the four source files in which each library subroutine (ADD, SUBTRACT, MULTIPLY, and DIVIDE) is a separate segment.
2. Assemble the programs, specifying a unique name for each program with the KEEP parameter in the ASSEMBLE command (or KEEP directive in source file). Start each program with a dummy segment that does not need to be in the library.

```
ASSEMBLE ADD KEEP=ADD
ASSEMBLE SUBTRACT KEEP=SUBTRACT
ASSEMBLE MULTIPLY KEEP=MULTIPLY
ASSEMBLE DIVIDE KEEP=DIVIDE
```

Each multi-segment program is saved as two object files, one with the extension .ROOT, and one with the extension .A.

3. Run the MAKELIB utility, specifying each object file to be included in the library file. For example, if you assembled the four files, creating the object files ADD.ROOT, ADD.A, SUBTRACT.ROOT, SUBTRACT.A, MULTIPLY.ROOT, MULTIPLY.A, DIVIDE.ROOT, DIVIDE.A and your library file is named ARITHLIB, then your command line should be as follows:

```
MAKELIB ARITHLIB +ADD.A +SUBTRACT.A +MULTIPLY.A +DIVIDE.A
```

The .ROOT files, which contained the dummy segments, do not need to be in the library.

4. Place the new library file in the LIBRARIES: subdirectory. (You can accomplish this in step 3 by specifying 13:ARITHLIB for the library file, or you can use the MOVE command after the file is created to directly copy the library file into the library prefix and delete ARITHLIB from the current prefix.)

Linker Script Files

The linker has a simple scripting mode that lets you create files to control the link process. These files can help you organize a project by placing the link information in a separate file.

Script files must be SRC files, just like source files for the compilers. The script files have a language stamp of LINKER.

The script file consists of comments, flags, file names, and keep names. At least one file name must appear in the script file; all other information is optional.

Comments include blank lines and any line that starts with a '*', '!' or ';' character.

The flags portion consists of any of the following flags.

```
+b +c +l +m +s +w +x
-b -c -l -m -s -w -x
```

All of these flags can also appear on the command line. They have the same meaning in the script file as they do when used with the LINK command. You can code duplicate flags, and you can use uppercase or lowercase letters. If you code a flag in the script file, and also use the same flag from the command line when you run the script, the command line flag will override the flag used in the script file. All flags are optional.

Flags are followed by object file names and library file names. You can use any legal GS/OS file name that does not include a space. You can use colons or slashes to indicate directories. You can specify the file names as a file name only, a partial path name, or a full path name. You must have at least one object file name. Basically, the file names are coded exactly like you would code them for the LINK command, except that you can use as many names as you like, and put them on separate lines in the script file.

The list of file names is followed by a keep name, which is optional. If a keep name is specified when you run the script, the keep name on the command line will override the keep name in the script file. The keep name is coded just like it is with the link command, as "keep=" followed by the file name.

Here's a sample script to show how some of these ideas can be put to use:

```
*
* Sample linker script
*

* Create a bank-relative program
  +b

* Don't express load the program
  -x

* Create a link map
  +l +s

* Link the program's object files
  main
  windows menus
  animation
  calc

* Use our special animation library
```

```
:mydisk:mylibs:animlib
```

```
* Save the resulting program  
keep=:mydisk:prog
```

To execute a linker script, use the `COMPILE`, `ASSEMBLE`, `ASML`, `CMPL`, `ASMLG`, `CMPLG` or `RUN` command. List the name of the linker script, along with any flags or keep names you wish to add. You can put the linker script on the same line as other source files, so long as it comes last. For example, assuming our sample script is called `linkit`, the following command will compile the main program, then link and execute it.

```
run main.asm linkit
```


Chapter 15

The Resource Compiler

This chapter describes the use and operation of the resource compiler. Key points covered in this chapter are:

- Creation of resource description files (Rez source files).
- Creating and using resource type statements.
- Using Rez to compile a resource description file to create a resource fork.
- Command, options, and capabilities of the resource compiler.

Overview

The Resource Compiler compiles a text file (or files) called a resource description file and produces a resource file as output. The resource decompiler, DeRez, decompiles an existing resource, producing a new resource description file that can be understood by the resource compiler.

Resource description files have a language type of REZ. By convention, the name of a resource description file ends with .rez. The REZ shell command enables you to set the language type to the rez language.

The resource compiler can combine resources or resource descriptions from a number of files into a single resource file. The resource compiler supports preprocessor directives that allow you to substitute macros, include other files, and use if-then-else constructs. (These are described under "Preprocessor Directives" later in this chapter.)

Resource Decompiler

The DeRez utility creates a textual representation of a resource file based on resource type declarations identical to those used by the resource compiler. (If you don't specify any type declarations, the output of DeRez takes the form of raw data statements.) The output of DeRez is a resource description file that may be used as input to the resource compiler. This file can be edited using the ORCA editor, allowing you to add comments, translate resource data to a foreign language, or specify conditional resource compilation by using the if-then-else structures of the preprocessor.

Type Declaration Files

The resource compiler and DeRez automatically look in the 13:RInclude directory, as well as the current directory, for files that are specified by file name on the command line. They also look in these directories for any files specified by a #include preprocessor directive in the resource description file.

Using the Resource Compiler and DeRez

The resource compiler and DeRez are primarily used to create and modify resource files. The resource compiler can also form an integral part of the process of building a program. For instance, when putting together a desk accessory or driver, you could use the resource compiler to combine the linker's output with other resources, creating an executable program file.

Structure of a Resource Description File

The resource description file consists of resource type declarations (which can be included from another file) followed by resource data for the declared types. Note that the resource compiler and resource decompiler have no built-in resource types. You need to define your own types or include the appropriate .rez files.

A resource description file may contain any number of these statements:

include	Include resources from another file.
read	Read the data fork of a file and include it as a resource.
data	Specify raw data.
type	Type declaration – declare resource type descriptions for subsequent <i>resource</i> statements.
resource	Data specification – specify data for a resource type declared in previous <i>type</i> statements.

Each of these statements is described in the sections that follow.

A type declaration provides the pattern for any associated resource data specifications by indicating data types, alignment, size and placement of strings, and so on. You can interspace type declarations and data in the resource description file so long as the declaration for a given resource precedes any resource statements that refer to it. An error is returned if data (that is, a *resource* statement) is given for a type that has not been previously defined. Whether a type was declared in a resource description file or in a #include file, you can redeclare it by providing a new declaration later in a resource description file.

A resource description file can also include comments and preprocessor directives. Comments can be included any place white space is allowed in a resource description file by putting them within the comment delimiters `/*` and `*/`. Note that comments do not nest. For example, this is one comment:

```
/* Hello /* there */
```

The resource compiler also supports the use of `//` as a comment delimiter. And characters that follow `//` are ignored, up to the end of the current line.

```
type 0x8001 { // the rest of this line is ignored
```

Preprocessor directives substitute macro definitions and include files, and provide if-then-else processing before other resource compiling takes place. The syntax of the preprocessor is very similar to that of the C-language preprocessor.

Sample Resource Description File

An easy way to learn about the resource description format is to decompile some existing resources. For example, the following command decompiles only the rIcon resources in an application called Sample, according to the declaration in 13:RInclude:Types.rez.

```
derez sample -only 0x8001 types.rez >derez.out
```

Note that DeRez automatically finds the file types.rez in 13:RInclude. After executing this command, the file derez.out would contain the following decompiled resource:

```
resource 0x8001 (0x1) {
    0x8000,
    20,
    28
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"FFFF FF00 0000 0000 0000 FFFF FFFF FFFF"
    $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
    $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
    $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
    $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
    $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
    $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
    $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
    $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
    $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
    $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
    $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
    $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
    $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
    $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
    $"0000 0000 0000 0000 0000 0000 0000 0000"
};
```

Note that this statement would be identical to the resource description in the original resource description file, with the possible exception of minor differences in formatting. The resource data corresponds to the following type declaration, contained in types.rez:

```
/*----- rIcon -----*/
type rIcon {
    hex integer; /* Icon Type bit 15 1 = color, 0 = mono */
image:
    integer = (Mask-Image)/8 - 6; /* size of icon data in bytes */
    integer; /* height of icon in pixels */
    integer; /* width of icon in pixels */
    hex string [$$Word(image)]; /* icon image */
mask:
    hex string; /* icon mask */
};
```

Type and resource statements are explained in detail in the reference section that follows.

Resource Description Statements

This section describes the syntax and use of the five types of resource description statements available for the resource compiler: `include`, `read`, `data`, `type` and `resource`.

Syntax Notation

The syntax notation in this chapter follows the conventions used earlier in the book. In addition, the following conventions are used:

- Words that are part of the resource description language are shown in the Courier font to distinguish them from surrounding text. The resource compiler is not sensitive to the case of these words.
- Punctuation characters such as commas (,), semicolons (;), and quotation marks (' and ") are to be written as shown. If one of the syntax notation characters (for example, [or]) must be written as a literal, it is shown enclosed by "curly" single quotation marks ('...'); for example,

```
bitstring ['length']
```

In this case, the brackets would be typed literally – they do *not* mean that the enclosed element is optional.

- Spaces between syntax elements, constants, and punctuation are optional they are shown for readability only.

Tokens in resource description statements may be separated by spaces, tabs, returns, or comments.

There are three terms used in the syntax of the resource description language that have not been used earlier to describe the shell. They are:

<i>resource-ID</i>	A long expression. (Expressions are defined later.)
<i>resource-type</i>	A word expression.
<i>ID-range</i>	A range of <i>resource-IDs</i> , as in <i>ID[:ID]</i> .

Include – Include Resources from Another File

The `include` statement lets you read resources from an existing file and include all or some of them.

An `include` statement can take the following forms:

- `include file [resource-type ['(' ID[:ID]')']] ;`

Read the resource of type *resource-type* with the specified resource ID range in *file*. If the resource ID is omitted, read all resources of the type *resource-type* in *file*. If *resource-type* is omitted, read all the resources in *file*.

- `include file not resource-type ;`

Read all resources in *file* that are not of the type *resource-type*.

- `include file resource-type1 as resource-type2 ;`

Read all resources of type *resource-type1* and include them as resources of *resource-type2*.

- `include file resource-type1 '('ID[:ID]')'`
`as resource-type2 '('ID[,attributes...]'');`

Read the resource in *file* of type *resource-type1* with the specified ID range, and include it as a resource of *resource-type2* with the specified ID. You can optionally specify resource attributes. (See "Resource Attributes," later in this section.)

Examples:

```
include "otherfile";           /* include all resources from the file */
include "otherfile" rIcon;     /* read only the rIcon resources */
include "otherfile" rIcon (128); /* read only rIcon resource 128 */
```

AS Resource Description Syntax

The following string variables can be used in the `as` resource description to modify the resource information in `include` statements:

<code>\$\$Type</code>	Type of resource from include file.
<code>\$\$ID</code>	ID of resource from include file.
<code>\$\$Attributes</code>	Attributes of resource from include file.

For example, to include all `rIcon` resources from one file and keep the same information but also set the preload attribute (64 sets it):

```
INCLUDE "file" rIcon (0:40) AS rIcon ($$ID, $$Attributes | 64);
```

The `$$Type`, `$$ID`, and `$$Attributes` variables are also set and legal within a normal resource statement. At any other time the values of these variables are undefined.

Resource Attributes

You can specify attributes as a numeric expression (as described in the *Apple IIGS Toolbox Reference*, Volume 3) or you can set them individually by specifying one of the keywords from any of the sets in Table 15.1. You can specify more than one attribute by separating the keywords with a comma (,).

<u>Default</u>	<u>Alternative</u>	<u>Meaning</u>
unlocked	locked	Locked resources cannot be moved by the Memory Manager.
moveable	fixed	Specifies whether the Memory Manager can move the block when it is unlocked.
nonconvert	convert	Convert resources require a resource converter.
handleload	absoluteload	Absolute forces the resource to be loaded at an absolute address.
nonpurgeable	purgeable1 purgeable2 purgeable3	Purgeable resources can be automatically purged by the Memory Manager. Purgeable3 are purged before purgeable2, which are purged before purgeable1.
unprotected	protected	Protected resources cannot be modified by the Resource Manager.
nonpreload	preload	Preloaded resources are placed in memory as soon as the Resource Manager opens the resource file.
crossbank	nocrossbank	A crossbank resource can cross memory bank boundaries. Only data, not code, can cross bank boundaries.
specialmemory	nospecialmemory	A special memory resource can be loaded in banks \$00, \$01, \$E0 and \$E1.
notpagealigned	pagealigned	A page-aligned resource must be loaded with a starting address that is an even multiple of 256.

Table 15.1 Resource Attribute Keywords

Read – Read Data as a Resource

```
read resource-type '(' ID [ , attributes ] ')' file ;
```

The `read` statement lets you read a file's data fork as a resource. It reads the data fork from *file* and writes it as a resource with the type *resource-type* and the resource ID *ID*, with the optional resource attributes.

Example:

```
read rText (0x1234, Purgeable3) "filename";
```

Data – Specify Raw Data

```
data resource-type '(' ID [ , attributes ] ')' '{'
  data-string
  '}' ;
```

Use the `data` statement to specify raw data as a sequence of bits, without any formatting.

The data found in *data-string* is read and written as a resource with the type *resource-type* and the ID *ID*. You can specify resource attributes.

When DeRez generates a resource description, it used the `data` statement to represent any resource type that doesn't have a corresponding type declaration or cannot be decompiled for some other reason.

Example:

```
data rPString (0xABCD) {
    $"03414243"
};
```

Type – Declare Resource Type

```
type resource-type [ '(' ID-range ')' ] '{'
    type-specification...
    '}' ;
```

A type declaration provides a template that defines the structure of the resource data for a single resource type or for individual resources. If more than one type declaration is given for a resource type the last one read before the data definition is the one that's used. This lets you override declarations from include files of previous resource description files.

After the type declaration, any resource statement for the type *resource-type* uses the declaration {*type-specification...*}. The optional *ID-range* specification causes the declaration to apply only to a given resource ID or range of IDs.

Type-specification is one or more of the following kinds of type specifier:

array	bitstring	boolean	byte	char
cstring	fill	integer	longint	point
pstring	rect	string	switch	wstring

You can also declare a resource type that uses another resource's type declaration by using the following variant of the type statement:

```
type resource-type1 [ '(' ID-range ')' ] as resource-type2 [ '(' ID ')' ] ;
```

Integer, Longint, Byte and Bitstring

```
[ unsigned ] [ radix ] integer [ = expression | symbol-definition ] ;
[ unsigned ] [ radix ] longint [ = expression | symbol-definition ] ;
[ unsigned ] [ radix ] byte [ = expression | symbol-definition ] ;
[ unsigned ] [ radix ] bitstring '[' length ']' [ = expression | symbol-definition ] ;
```

In each case, space is reserved in the resource for an integer or a long integer.

If the type appears alone, with no other parameters, the resource compiler sets aside space for a value that must be given later when the resource type is used to define an actual resource.

A type followed by a equal sign and an expression defines a value that will be preset to some specific integer. Since the value is already given, you do not need to code the value again when the resource type is used to define a resource.

A symbol-definition is an identifier, an equal sign, and an expression, optionally followed by a comma and another symbol definition. It sets up predefined identifier that can be used to fill in the value. You still have the option of coding a numeric value, or you can use one of the constants. This is not a default value, though; you still must code either one of the constants or a numeric value when you use the resource type to define a resource.

The *unsigned* prefix signals DeRez that the number should be displayed without a sign – that the high-order bit can be used for data and the value of the integer cannot be negative. The

Shell Reference Manual

unsigned prefix is ignored by the resource compiler but is needed by DeRez to correctly represent a decompiled number. The resource compiler uses a sign if it is specified in the data. For example, \$FFFFFF85 and -\$7B are equivalent.

Radix is one of the following constants:

hex decimal octal binary literal

The radix is used by DeRez to decide what number format to use for the output. The radix field is ignored by the resource compiler.

Each of the numeric types generates a different format of integer. In each case, the value is in two's complement form, least significant byte first. The various formats are:

<u>type</u>	<u>size</u>	<u>range</u>
byte	1	-128..255
integer	2	-32768..65535
longint	4	-2147483648..4294967295
bitstring[length]	varies	varies

Sizes are in bytes. The range may seem a little odd at first; the resource compiler accepts either negative or positive values, treating positive values that would normally be too large for a signed value of the given length as if the value were unsigned.

The bitstring type is different from most types in other languages. It is a variable-length integer field, where you specify the number of bits you want as the length field. If you specify a value that only fills part of a byte, then the next field will pick up where the bitstring field stopped. For example, two bitstring[4] values, placed back to back, would require only one byte of storage in the resource file. In general, you should be sure that bitstring fields end on even byte values so the following fields don't get bit aligned to the end of the partially filled byte.

Example:

```
/*----- rToolStartup -----*/
type rToolStartup {
    integer = 0; /* flags must be zero */
    Integer mode320 = 0, mode640 = $80; /* mode to start quickdraw */
    Integer = 0;
    Longint = 0;
    integer = $$Countof(TOOLRECS); /* number of tools */
    array TOOLRECS {
        Integer; /* ToolNumber */
        Integer; /* version */
    };
};

resource rToolStartup (1) {
    mode640,
    {
        1,1, /* Tool Locator */
        2,1, /* Memory Manager */
        3,1, /* Miscellaneous Tool Set */
        4,1, /* QuickDraw II */
    }
}
```

```

5,1,      /* Desk Manager */
6,1,      /* Event Manager */
11,1,     /* Integer Math Tool Set */
14,1,     /* Window Manager */
15,1,     /* Menu Manager */
16,1,     /* Control Manager */
18,1,     /* QuickDraw II Auxiliary */
20,1,     /* LineEdit Tool Set */
21,1,     /* Dialog Manager */
22,1,     /* Scrap Manager */
27,1,     /* Font Manager */
28,1,     /* List Manager */
30,1,     /* Resource Manager */
    }
};

```

Boolean

```
boolean [= constant | symbolic-value... ] ;
```

A boolean value is a one-bit value, set to either false (0) or true (1). You can also use the numeric values.

True and false are actually predefined constants.

The type boolean is equivalent to

```
unsigned bitstring[1]
```

Example:

```

type 0x001 {
    boolean;
    boolean;
    boolean;
    boolean;
    bitstring[4] = 0;
};

resource 0x001 (1) {
    true, false, 0, 1
};

```

Character

```
char [= string | symbolic-value... ] ;
```

A character value is an 8-bit value which holds a one-character string. It is equivalent to string[1].

Example:

```

/*----- rMenuItem -----*/
type rMenuItem {

```

Shell Reference Manual

```
integer = 0; /* version must be zero */
integer; /* item ID */
char; /* item char */
char; /* alt char */
integer; /* item check */
integer; /* flags */
longint; /* item titleref */
};

resource rMenuItem (1) {
    256,
    "Q", "q",
    0,
    0,
    1
};
```

String, PString, WString and CString

string-type ['[' *length* ']'] [= *string* | *symbol-value*...] ;

String types are used to define a string in one of four formats. The format of the string is determined by selecting one of the following for *string-type*:

[hex] string	Plain string; no length indicator or terminal character is generated. The optional hex prefix tells DeRez to display it as a hexadecimal string. <code>String[n]</code> contains <i>n</i> characters and is <i>n</i> bytes long. The type <code>char</code> is a shorthand for <code>string[1]</code> .
pstring	Pascal string; a leading byte containing the number of characters in the string is generated. <code>Pstring[n]</code> contains <i>n</i> characters and is <i>n</i> +1 bytes long. Since the length must fit in a byte value, the maximum length of a pstring is 255 characters. If the string is too long, a warning is given and the string is truncated.
wstring	Word string; this is a very large pstring. The length of a wstring is stored in a two-byte field, giving a maximum length of 65535 characters. <code>Pstring[n]</code> contains <i>n</i> characters and is <i>n</i> +2 bytes long. The order of the bytes in the length word is least significant byte first; this is the normal order for bytes on the Apple IIGS.
cstring	C string; a trailing null byte is added to the end of the characters. <code>Cstring[n]</code> contains <i>n</i> -1 characters and is <i>n</i> bytes long. A C string of length 1 can be assigned only the value "", since <code>cstring[1]</code> only has room for the terminating null.

Each string type can be followed by an optional *length* indicator in brackets. *length* is an expression indicating the string length in bytes. *length* is a positive number in the range 1..2147483647 for string and cstring, in the range 1..255 for pstring, and in the range 1..65535 for wstring.

If no length indicator is given, a pstring, wstring or cstring stores the number of characters in the corresponding data definition. If a length indicator is given, the data may be truncated on the right or padded on the right. The padding characters for all strings are nulls. If the data contains

more characters than the length indicator provides for, the string is truncated and a warning message is given.

Examples:

```
/*----- rPString -----*/
type rPString {
    pstring;          /* String */
};

/*----- rCString -----*/
type rCString {
    cstring;          /* String */
};

/*----- rWString -----*/
type rWString {
    wstring;          /* String */
};

/*----- rErrorString -----*/
type rErrorString {
    string;
};

resource rPString (1) {
    "p-string",
};

resource rCString (1) {
    "c-string",
};

resource rWString (1) {
    "GS/OS input string",
};

resource rErrorString (1) {
    "Oops",
};
```

Point and Rectangle

```
point [ = point-constant | symbolic-value... ] ;
rect [ = rect-constant | symbolic-value... ] ;
```

Because points and rectangles appear so frequently in resource files, they have their own simplified syntax. In the syntax shown, a point-constant is defined like this:

```
{ ' x-integer-expression ' , ' y-integer-expression ' }
```

while a rect-constant looks like this:

{ 'integer-expression' , 'integer-expression' , 'integer-expression' , 'integer-expression' }

A point type creates a pair of integer values, with the first value corresponding to the horizontal point value and the second to the vertical point value. A rect type is a pair of points, with the top left corner of the rectangle specified first, followed by the bottom right corner.

Example:

```
/*----- rWindParam1 -----*/
type rWindParam1 {
    integer = $50;                /*length of parameter list, should be $50*/
    integer;                      /* wFrameBits */
    longint;                      /* wTitle */
    longint;                      /* wRefCon */
    rect;                         /* ZoomRect */
    longint;                      /* wColor ID */
    point;                        /* Origin */
    point;                        /* data size */
    point;                        /* max height-width */
    point;                        /* scroll ver hors */
    point;                        /* page vers horiz */
    longint;                      /* winfoRefcon */
    integer;                      /* wInfoHeight */
    fill long[3];                 /* wFrameDefProc,wInfoDefProc,wContDefProc */
}

    rect;                        /* wposition */
    longint behind=0,infront=-1; /* wPlane */
    longint;                      /* wStorage */
    integer;                      /* wInVerb */
};

resource rWindParam1 (1) {
    0x80E4,                      /* wFrameBits */
    1,                          /* wTitle */
    0,                          /* wRefCon */
    {0,0,0,0},                  /* ZoomRect */
    0,                          /* wColor ID */
    {0,0},                      /* Origin */
    {416,160},                  /* data size */
    {416,160},                  /* max height-width */
    {0,0},                      /* scroll ver hors */
    {0,0},                      /* page vers horiz */
    0,                          /* winfoRefcon */
    0,                          /* wInfoHeight */
    {32,32,448,192},            /* wposition */
    infront,                    /* wPlane */
    0,                          /* wStorage */
    0x0200                      /* wInVerb */
};
```


Fill

```
fill fill-size [ '[' length '[' ] ;
```

The resource created by a resource definition has no implicit alignment. It's treated as a bit stream, and integers and strings can start at any bit. The fill specifier is a way of padding fields so that they begin on a boundary that corresponds to the field type.

The fill statement causes the resource compiler to add the specified number of bits to the data stream. The bits added are always set to 0. *fill-size* is one of the following:

```
bit      nibble byte   word   long
```

These declare a fill of 1, 4, 8, 16 or 32 bits, respectively. Any of these can be followed by a *length* modifier. *length* can be any value up to 2147483647; it specifies the number of these bit fields to insert. For example, all of the following are equivalent:

```
fill word[2];
fill long;
fill bit[32];
```

Fill statements are sometimes used as place holders, filling in constant values of zero. You can see an example of the fill statement used for this purpose in the `rWindParam1` resource type defined in `types.rez`. The example in the last section shows this resource type in use.

Array

```
[ wide ] array [ array-name [ '[' length '[' ] [ '{' array-list '}' ] ;
```

The *array-list* is a list of type specifications. It can be repeated zero or more times. The *wide* option outputs the array data in a wide display format when the resource is decompiled with DeRez; this causes the elements that make up the *array-list* to be separated by a comma and space instead of a comma, return, and tab.

Either *array-name* or *[length]* may be specified. *Array-name* is an identifier. If the array is named, then a preceding statement should refer to that array in a constant expression with the `$$countof(array-name)` function, otherwise DeRez will treat the array as an open-ended array. For example,

```
type rToolStartup {
    integer = 0;                                /* flags must be zero */
    Integer mode320 = 0, mode640 = $80; /* mode to start quickdraw */
    Integer = 0;
    Longint = 0;
    integer = $$Countof(TOOLRECS);             /* number of tools */
        array TOOLRECS {
            Integer;                             /* ToolNumber */
            Integer;                             /* version */
        };
};
```

The `$$countof(array-name)` function returns the number of array elements (in this case, the number of tool number, version pairs) from the resource data.

If *length* is specified, there must be exactly *length* elements.

Array elements are generated by commas. Commas are element separators. Semicolons are element terminators.

For an example of an `rToolStartup` resource, see "Integer, Longint, Byte and Bitstream," earlier in this chapter.

Switch

```
switch '{' case-statement... '}' ;
```

The switch statement lets you select one of a variety of types when you create your resource. Each of the types within the switch statement are placed on a case label, which has this format:

```
case case-name : [case-body ; ] ...
```

Case-name is an identifier. *Case-body* may contain any number of type specifications and must include a single constant declaration per case, in this form:

```
key data-type = constant
```

The key value determines which case applies. For example,

```
/*----- rControlTemplate -----*/
type rControlTemplate {
    integer = 3+$$optionalcount (Fields); /* pCount must be at least 6 */
    longint;                               /* Application defined ID */
    rect;                                  /* controls bounding rectangle */
    switch {

        case SimpleButtonControl:
            key longint = 0x80000000; /* procRef */
            optional Fields {
                integer;                /* flags */
                integer;                /* more flags */
                longint;                /* refcon */
                longint;                /* Title Ref */
                longint;                /* color table ref */
                KeyEquiv;
            };

        case CheckControl:
            key longint = 0x82000000; /* procRef */
            optional Fields {
                integer;                /* flags */
                integer;                /* more flags */
                longint;                /* refcon */
                longint;                /* Title Ref */
                integer;                /* initial value */
                longint;                /* color table ref */
                KeyEquiv;
            };
        ...and so on.
```

```
};
```

Symbol Definitions

Symbolic names for data type fields simplify the reading and writing of resource definitions. Symbol definitions have the form

```
name = value [, name = value ]...
```

The “= *value*” part of the statement can be omitted for numeric data. If a sequence of values consists of consecutive numbers, the explicit assignment can be left out; if *value* is omitted, it is assumed to be 1 greater than the previous value. (The value is assumed to be 0 if it is the first value in the list.) This is true for bitstrings (and their derivatives, byte, integer, and longint). For example,

```
integer Emily, Kelly, Taylor, Evan, Trevor, Sparkle=8;
```

In this example, the symbolic names Emily, Kelly, Taylor, Evan, and Trevor are automatically assigned the numeric values 0, 1, 2, 3, and 4.

Memory is the only limit to the number of symbolic values that can be declared for a single field. There is also no limit to the number of names you can assign to a given value; for example,

```
integer    Emily=0, Kelly=1, Taylor=2, Evan=3,
           Trevor=16, Sparkle=0, Twinkle=1, Raphael=2,
           Michaelangelo=3, Nagel=16;
```

Delete – Delete a Resource

```
delete resource-type [ '(' ID [ : ID ] ')' ] ;
```

This statement deletes the resource of *resource-type* with the specified ID or ID range from the resource compiler output file. If ID or ID range is omitted, all resources of *resource-type* are deleted.

The delete function is valid only if you specify the `-a` (append) option on the resource compiler command line. (It wouldn't make sense to delete a resource while creating a new resource file from scratch.)

You can delete resources that have their protected bit set only if you use the `-ov` option on the resource compiler command line.

Change – Change a Resource's Vital Information

```
change resource-type1 [ '(' ID [ : ID ] ')' ]
      resource-type2 '(' ID [ , attributes... ] ')' ;
```

This statement changes the resource of *resource-type1* with the specified ID or ID range in the resource compiler output file to a resource of *resource-type2* and the specified ID. If ID or ID range is omitted, all resources of *resource-type1* are changed.

The change function is valid only if you specify the `-a` (append) option on the resource compiler command line. (It wouldn't make sense to change resources while creating a new resource file from scratch.)

Resource – Specify Resource Data

```
resource resource-type '(' ID [ , attributes ] ')' '{'
    [ data-statement [ , data-statement ] ... ]
    '}'
```

Resource statements specify actual resources, based on previous type declarations.

This statement specifies the data for a resource of type *resource-type* and ID *ID*. The latest type declaration declared for *resource-type* is used to parse the data specification.

Data statements specify the actual data; data-statements appropriate to each resource type are defined in the next section.

The resource definition generates an actual resource. A resource statement can appear anywhere in the resource description file, or even in a separate file specified on the command line or as an `#include` file, as long as it comes after the relevant type declaration.

For examples of resource statements, see the examples following the various data statement types, earlier in this chapter.

Data Statements

The body of the data specification contains one data statement for each declaration in the corresponding type declaration. The base type must match the declaration.

<u>Base type</u>	<u>Instance types</u>
string	string, cstring, pstring, wstring, char
bitstring	boolean, byte, integer, longint, bitstring
rect	rect
point	point

Switch data

Switch data statements are specified by using this format:

```
switch-name data-body
```

For example, the following could be specified for the `rControlTemplate` type used in an earlier example:

```
CheckControl { enabled, "Check here" },
```

Array data

Array data statements have this format:

```
'{' [ array-element [ , array-element ] ... ] '}'
```

where an *array-element* consists of any number of data statements separated by commas.

For example, the following data might be given for the `rStringList` resource (the type is shown so you won't have to refer to `types.rez`, where it is defined):

```
type rStringList {
    integer = $$Countof(StringArray);
    array StringArray {
        pstring;          /* String          */
    };
};

resource rStringList (280) {
    {
        "this",
        "is",
        "a",
        "test"
    }
};
```

Sample resource definition

This section describes a sample resource description file for an icon. (See the Apple IIGS Toolbox Reference, Volume 3 for information about resource icons.) The type statement is included for clarity, but would normally be included using an include statement.

```
type rIcon {
    hex integer;          /* icon type bit 15  1 = color,
                        0 = mono */
    image:
        integer = (Mask-Image)/8 - 6; /* size of icon data in bytes */
        integer;          /* height of icon in pixels */
        integer;          /* width of icon in pixels */
        hex string [$$Word(image)]; /* icon image */
    mask:
        hex string;      /* icon mask */
};

resource rIcon (1) {
    0x8000,                /* Kind */
    9,                    /* Height */
    32,                   /* Width */
    $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
    $"FFFFFF0000000000000000000000000000"
    $"FFFFFF00FFFFFFFFFFFF00FFFFFFFFFFFF"
    $"FFFFFF00FFFFFFFFFFFF00FFFFFFFFFFFF"
    $"FFFFFF00FFFFFFFFFFFF00FFFFFFFFFFFF"
    $"FFFFFF00FFFFFFFFFFFF00FFFFFFFFFFFF"
    $"FFFFFF00FFFFFFFFFFFF00FFFFFFFFFFFF"
    $"FFFFFF0000000000000000000000000000"
    $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"

    $"0000000000000000000000000000000000"
```

```

$"0000FFFFFFFFFFFFFFFF00000000000000"
$"0000FFFFFFFFFFFFFFFF00000000000000"
$"0000FFFFFFFFFFFFFFFF00000000000000"
$"0000FFFFFFFFFFFFFFFF00000000000000"
$"0000FFFFFFFFFFFFFFFF00000000000000"
$"0000FFFFFFFFFFFFFFFF00000000000000"
$"0000FFFFFFFFFFFFFFFF00000000000000"
$"00000000000000000000000000000000"
};

```

This data definition declares a resource of type `rlcon`, using whatever type declaration was previously specified for `rlcon`. The 8 in the resource type specification (0x8000) identifies this as a color icon.

The icon is 9 pixels high by 32 pixels wide.

The specification of the icon includes a pixel image and a pixel mask.

Labels

Labels support the more complicated resources. Use labels within a resource type declaration to calculate offsets and permit accessing of data at the labels. The `rlcon` resource, for example, uses labels to specify the pixel image and mask of the icon.

The syntax for a label is:

```

label ::=          character {alphanum}* ':'
character ::=      '_' | A | B | C ...
alphanum ::=       character | number
number ::=         0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Labeled statements are valid only within a resource type declaration. Labels are local to each type declaration. More than one label can appear on a statement.

Labels may be used in expressions. In expressions, use only the identifier portion of the label (that is, everything up to, but excluding, the colon). See “Declaring Labels Within Arrays” later in this chapter for more information.

The value of a label is always the offset, in bits, between the beginning of the resource and the position where the label occurs when mapped to the resource data. In this example,

```

type 0xC000 {
    cstring;
endOfString:
    integer = endOfString;
};

resource 0xC000 (8) {
    "Neato"
}

```

the integer following the `cstring` would contain:

```
( len("Neato") [5] + null byte [1] ) * 8 [bits per byte] = 48.
```

Built-in Functions to Access Resource Data

In some cases, it is desirable to access the actual resource data to which a label points. Several built-in functions allow access to that data:

- `$$BitField (label, startingPosition, numberOfBits)`

Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

- `$$Byte (label)`

Returns the byte found at *label*.

- `$$Word (label)`

Returns the word found at *label*.

- `$$Long (label)`

Returns the long word found at *label*.

For example, the resource type `rPString` could be redefined without using a pstring. Here is the definition of `rPString` from `Types.rez`:

```
type rPString {
    pstring;
};
```

Here is a redefinition of `rPString` using labels:

```
type rPString {
    len: byte = (stop - len) / 8 - 1;
        string[$$Byte(len)];
    stop: ;
};
```

Declaring Labels Within Arrays

Labels declared within arrays may have many values. For every element in the array there is a corresponding value for each label defined within the array. Use array subscripts to access the individual values of these labels. The subscript values range from 1 to *n* where *n* is the number of elements in the array. Labels within arrays that are nested in other arrays require multidimensional subscripts. Each level of nesting adds another subscript. The rightmost subscript varies most quickly. Here is an example:

```
type 0xFF01 {
    integer = $$CountOf(array1);
    array array1 {
        integer = $$CountOf(array2);
```

```

        array array2 {
foo:          integer;
        };
    };
resource 0xFF01 (128) {
    {
        {1,2,3},
        {4,5}
    }
};

```

In the example just given, the label `foo` takes on these values:

```

foo[1,1] = 32      $$Word(foo[1,1]) = 1
foo[1,2] = 48      $$Word(foo[1,2]) = 2
foo[1,3] = 64      $$Word(foo[1,3]) = 3
foo[2,1] = 96      $$Word(foo[2,1]) = 4
foo[2,2] = 112     $$Word(foo[2,2]) = 5

```

Another built-in function may be helpful in using labels within arrays:

```
$$ArrayIndex(arrayname)
```

This function returns the current array index of the array *arrayname*. An error occurs if this function is used anywhere outside the scope of the array *arrayname*.

Label Limitations

Keep in mind the fact that the resource compiler and DeRez are basically one-pass compilers. This will help you understand some of the limitations of labels.

To decompile a given type, that type must not contain any expressions with more than one undefined label. An undefined label is a label that occurs lexically after the expression. To define a label, use it in an expression before the label is defined.

This example demonstrates how expressions can have only one undefined label:

```

type 0xFF01 {
    /* In the expression below, start is defined, next is undefined.*/
start:    integer = next - start;
    /* In the expression below, next is defined because it was used
        in a previous expression, but final is undefined.*/
middle:   integer = final - next;
next:     integer;
final:
};

```

Actually, the resource compiler can compile types that have expressions containing more than one undefined label, but the DeRez cannot decompile those resources and simply generates data resource statements.

The label specified in `$$BitField()`, `$$Byte()`, `$$Word()`, and `$$Long()` must occur lexically before the expression; otherwise, an error is generated.

An Example Using Labels

In the following example, the definition for the `rIcon` resource uses the labels `image` and `mask`.

```
type rIcon {
    hex integer;          /* Icon Type bit 15  1 = color, 0 = mono */
image:
    integer = (Mask-Image)/8 - 6; /* size of icon data in bytes */
    integer;              /* height of icon in pixels */
    integer;              /* width of icon in pixels */
    hex string [$$Word(image)]; /* icon image */
mask:
    hex string;           /* icon mask */
};
```

In the data corresponding to that definition, pixel images are provided for the `image` and `mask`.

```
resource rIcon (1) {
    0x8000,                /* Kind */
    9,                     /* Height */
    32                     /* Width */
    $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
    $"FFFFFF0000000000000000FFFFFFFF"
    $"FFFFFF00FFFFFFFF00FFFFFFFF"
    $"FFFFFF00FFFFFFFF00FFFFFFFF"
    $"FFFFFF00FFFFFFFF00FFFFFFFF"
    $"FFFFFF00FFFFFFFF00FFFFFFFF"
    $"FFFFFF00FFFFFFFF00FFFFFFFF"
    $"FFFFFF0000000000000000FFFFFFFF"
    $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"

    $"000000000000000000000000000000"
    $"0000FFFFFFFFFFFFFFFF000000000000"
    $"0000FFFFFFFFFFFFFFFF000000000000"
    $"0000FFFFFFFFFFFFFFFF000000000000"
    $"0000FFFFFFFFFFFFFFFF000000000000"
    $"0000FFFFFFFFFFFFFFFF000000000000"
    $"0000FFFFFFFFFFFFFFFF000000000000"
    $"0000FFFFFFFFFFFFFFFF000000000000"
    $"000000000000000000000000000000"
};
```

Preprocessor Directives

Preprocessor directives substitute macro definitions and include files and provide if-then-else processing before other resource compiler processing takes place.

The syntax of the preprocessor is very similar to that of the C-language preprocessor. Preprocessor directives must observe these rules and restrictions:

- Each preprocessor statement must begin on a new line, be expressed on a single line, and be terminated by a return character.

- The pound sign (#) must be the first character on the line of a preprocessor statement (except for spaces and tabs).
- Identifiers (used in macro names) may be letters (A–Z, a–z), digits (0–9), or the underscore character (_).
- Identifiers may be any length.
- Identifiers may not start with a digit.
- Identifiers are not case sensitive.

Variable Definitions

The #define and #undef directives let you assign values to identifiers:

```
#define macro data
#undef macro
```

The #define directive causes any occurrence of the identifier *macro* to be replaced with the text *data*. You can extend a macro over several lines by ending the line with the backslash character (\), which functions as the resource compiler's escape character. Here is an example:

```
#define poem "I wander \
thro\' each \
charter\'d street"
```

Quotation marks within strings must also be escaped. See "Escape Characters: later in this chapter for more information about escape characters.

The #undef directive removes the previously defined identifier macro. Macro definitions can also be removed with the -undef option on the resource compiler command line.

The following predefined macros are provided:

<u>Variable</u>	<u>Value</u>
true	1
false	0
rez	1 or 0 (1 if the resource compiler is running, 0 if DeRez is running)
derez	1 or 0 (0 if the resource compiler is running, 1 if DeRez is running)

If-Then-Else Processing

These directives provide conditional processing:

```
#if expression
[ #elif expression ]
[ #else ]
#endif
```

Expression is defined later in this chapter.

When used with the #if and #elif directives, *expression* may also include one of these terms:

```
defined identifier
defined '(' identifier ')'
```

The following may also be used in place of `#if`:

```
#ifdef macro
#ifndef macro
```

For example,

```
#define Thai
Resource rPstring (199) {
#ifdef English
    "Hello"
#elif defined (French)
    "Bonjour"
#elif defined (Thai)
    "Sawati"
#elif defined (Japanese)
    "Konnichiwa"
#endif
};
```

Printf Directive

The `#printf` directive is provided to aid in debugging resource description files. It has the form

```
#printf(formatString, arguments...)
```

The format of the `#printf` statement is exactly the same as that of the `printf` statement in the C language, with one exception: There can be no more than 20 arguments. This is the same restriction that applies to the `$$format` function. The `#printf` directive writes its output to diagnostic output. Note that the `#printf` directive does not end with a semicolon.

Here's an example:

```
#define          Tuesday          3
#ifdef Monday
#printf("The day is Monday, day #%d\n", Monday)
#elif defined(Tuesday)
#printf("The day is Tuesday, day #%d\n", Tuesday)
#elif defined(Wednesday)
#printf("The day is Wednesday, day #%d\n", Wednesday)
#elif defined(Thursday)
#printf("The day is Thursday, day #%d\n", Thursday)
#else
#printf("DON'T KNOW WHAT DAY IT IS!\n")
#endif
```

The file just listed generates this text:

```
The day is Tuesday, day #3
```

Formatstring is a text string which is written more or less as is to error out. There are two cases when the string is not written exactly as typed: escape characters and conversion specifiers.

Escape sequences are used to encode characters that would not normally be allowed in a string. The examples show the most commonly used escape sequence, `\n`. The `\` character marks the beginning of an escape sequence, telling the resource compiler that the next character is special. In this case, the next character is `n`, which indicates a newline character. Printing `\n` is equivalent to a `writeln` in Pascal or a `PutCR` macro from assembly language. For a complete description of escape sequences, see "Escape Characters," later in this chapter.

Conversion Specifiers

Conversion specifiers are special sequences of characters that define how a particular value is to be printed. While the resource compiler actually accepts all of the conversion specifiers allowed by the C language (it is written in C, and uses C's `sprintf` function to format the string for this statement), many of the conversion specifiers that are used by C are not useful in the resource compiler, and some of the others are not commonly used. For example, technically the resource compiler supports floating-point output, but it does not have a floating point variable type, so the conversion specifiers for floating point values are not of much use. Only those conversion specifiers that are generally used in the resource compiler will be covered here.

Each conversion specifier starts with a `%` character; to write a `%` character, code it twice, like this:

```
printf("100%%\n");
```

Conversion specifiers are generally used to write string or numeric arguments. For example, the `%n` conversion specifier is used to write a two-byte integer. You can put one of several characters between the `%` characters that starts a conversion specifier and the letter character that indicates the type of the argument; each of these additional characters modifies the format specifier in some way. The complete syntax for a format specifier is

`% flag [field-width] [size-specifier] conversion`

Flag is one or more of the characters `-`, `0`, `+` or a space. The entire field is optional. These flags effect the way the output is formatted:

- If a formatted value is shorter than the minimum field width, it is normally right-justified in the field by adding characters to the left of the formatted value. If the `-` flag is used, the value is left-justified.
- 0 If a formatted value is shorter than the minimum field width, it is normally padded with space characters. If the `0` flag is used, the field is padded with zeros instead of spaces. The `0` pad character is not used if the value is left-justified.
- + Forces signed output, adding a `+` character before positive integers.
- space Adds a space before positive numbers (instead of a `+`) so they line up with columnated negative numbers.

Field-width gives the number of characters to use for the output field. If the number of characters needed to represent a value is less than the field width, spaces are added on the left to fill out the field. For example, the statement

```
printf("%10n%10n\n", a, b);
```

could be used to print two columns of numbers, where each column is ten characters wide and the numbers are right-justified.

The *size-specifier* gives the size of the operand. If the *size-specifier* is omitted, the resource compiler expects to find an integer parameter in the parameter list when it processes any of the numeric conversion specifiers. If the size specifier is h, a byte is expected, while l indicates that the resource compiler should look for a longint value.

Conversion tells what size and type of operand to expect and how to format the operand:

<u>Conversion</u>	<u>Format</u>
d	signed integer
u	unsigned integer
o	unsigned octal integer
x	unsigned hexadecimal number; lowercase letters are used
X	unsigned hexadecimal number; uppercase letters are used
c	character
s	c-string
p	p-string
%	write a single % character

You must include exactly one parameter after the format string for each conversion specifier in the format string, and the types of the parameters must agree exactly with the types indicated by the conversion specifiers. Parameters are matched with conversion specifiers on a left-to-right basis.

Include Directive

The #include directive reads a text file:

```
#include "filename"
```

The directive behaves as if all of the lines in *file* were placed in the current source file, replacing the line with the directive. The maximum nesting is to ten levels. For example,

```
#include ($$Shell("ORCA")) "MyProject MyTypes.rez"
```

Note that the #include preprocessor directive (which includes a file) is different from the previously described include statement, which copies resources from another file.

The #include directive will look up to three places for the file, in order:

1. The current directory.
2. The directory where the source file is located (generally the current directory, but not always).
3. The directory 13:RInclude.

Append Directive

This directive allows you to specify additional files to be compiled by the resource compiler. The format is:

```
#append filename
```

This directive must appear between resource or type statements. The *filename* variable is the name of the next file to be compiled. The same search rules apply here that apply to the #include directive. Normally you should place this directive at the end of a file because everything after it is ignored. Do not place a #append directive in an include file.

If you use more than one #append directive, the order in which you put them is important. When the resource compiler sees an #append directive, it checks the language type of the appended file. If it is the same language, that is, REZ, the effect is the same as if the files had been concatenated into a single file. If they are in different languages, the shell quits the resource compiler and begins a new assembly or compilation. Two examples will illustrate why the order is important.

In the first example, suppose you have the following three files, each appended to the preceding file.

```
file1.rez
file2.rez
file3.asm
```

The Compile command calls the resource compiler to process file1.rez because the language is REZ. When the resource compiler encounters the #append directive for file2.rez it continues processing as if file1.rez and file2.rez had been concatenated into a single file. When it encounters the #append directive for file3.asm, the resource compiler finishes processing and returns control to the shell which calls the assembler to assemble file3.asm.

The result is different if the order of the files is changed, as follows:

```
file1.rez
file3.asm
file2.rez
```

The resource compiler processes file1.rez. When it encounters the #append directive for file3.asm, the resource compiler finishes processing and returns control to the ORCA shell because the language stamp is different. The shell calls the assembler to process file3.asm. When the assembler is finished processing, it returns control to the shell which calls the resource compiler to process file2.rez. However, since this is a separate compilation from that of file1.rez, the resource compiler knows nothing about symbols from file1.rez when compiling file2.rez.

DeRez handles #append directives differently from the resource compiler. For DeRez the file being appended must have a language stamp of REZ or DeRez will treat the #append directive as an end-of-file marker. DeRez will not return control to the shell after finishing processing. Therefore, in the previous example, DeRez would process file1.rez only and then finish processing.

Resource Description Syntax

This section describes the details of the resource description syntax.

Numbers and Literals

All arithmetic is performed as 32-bit signed arithmetic. The basic formats are shown in Table 15.2.

<u>Numeric Type</u>	<u>Form</u>	<u>Meaning</u>
Decimal	nnn...	Signed decimal constant between 2,147,483,647 and -2,147,483,648. Do not use a leading zero. (See octal.)
Hexadecimal	0Xhhh...	Signed hexadecimal constant between 0X7FFFFFFF and 0X80000000.
	\$hhh...	Alternate form for hexadecimal constants.
Octal	0ooo...	Signed octal constant between 01777777777 and 02000000000. A leading zero indicates that the number is octal.
Binary	0Bbbb...	Signed binary constant between 0B11111111111111111111111111111111 and 0B10000000000000000000000000000000.
Literal	'aaaa'	One to four printable ASCII characters or escape characters. If there are fewer than four characters in the literal, the characters to the left (high bits) are assumed to be \$00. Characters that are not in the printable character set, and are not the characters \ and \\ (which have special meanings), can be escaped according to the character escape rules. (See “Strings” later in this section.)

Table 15.2: Numeric Constants

Literals and numbers are treated in the same way by the resource compiler. A literal is a value within single quotation marks; for instance, 'A' is a number with the value 65; on the other hand, "A" is the character A expressed as a string. Both are represented in memory by the bitstring 01000001. (Note, however, that "A" is not a valid number and 'A' is not a valid string.) The following numeric expressions are all equivalent:

'B' 66 'A'+1

Literals are padded with nulls on the left side so that the literal 'ABC' is stored as shown in Figure 15.3.

'ABC' =

\$00	A	B	C
------	---	---	---

Figure 15.3: Padding of literals

Expressions

An expression may consist of simply a number or a literal. Expressions may also include numeric variables, labels, and system functions.

Table 15.3 lists the operators in order of precedence with highest precedence first – groupings indicate equal precedence. Evaluation is always left to right when the priority is the same.

<u>Precedence</u>	<u>Operator</u>	<u>Meaning</u>
1.	(expr)	Forced precedence in expression calculation
2.	-expr	Arithmetic (two's complement) negation of expr
	~expr	Bitwise (one's complement) negation of expr
	!expr	Logical negation of expr
3.	expr1 * expr2	Multiplication
	expr1 / expr2	Integer division
	expr1 % expr2	Remainder from dividing expr1 by expr2
4.	expr1 + expr2	Addition
	expr1 - expr2	Subtraction
5.	expr1 << expr2	Shift left; shift expr1 left by expr2 bits
	expr1 >> expr2	Shift right; shift expr1 right by expr2 bits
6.	expr1 > expr2	Greater than
	expr1 >= expr2	Greater than or equal to
	expr1 < expr2	Less than
	expr1 <= expr2	Less than or equal to
7.	expr1 == expr2	Equal
	expr1 != expr2	Not equal
8.	expr1 & expr2	Bitwise AND
9.	expr1 ^ expr2	Bitwise XOR
10.	expr1 expr2	Bitwise OR
11.	expr1 && expr2	Logical AND
12.	expr1 expr2	Logical OR

Table 15.3: Resource Description Operators

The logical operators `!`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `&&`, and `||` evaluate to 1 (true) or 0 (false).

Variables and Functions

There are several predefined variables that are preset by the resource compiler, or that take on specific meaning based on how they are used in your resource description file. Some of these resource compiler variables also contain commonly used values. All Rez variables start with `$$` followed by an alphanumeric identifier.

The following variables and functions have string values:

`$$Date` Current date. It is useful for putting time-stamps into the resource file. The format of the string is: weekday, month dd, yyyy. For example, August 10, 1989.

`$$Format("formatString" , arguments)`
Works just like the `#printf` directive except that `$$Format` returns a string rather than printing to standard output. (See “Print Directive” earlier in this chapter.)

`$$Resource("filename" , 'type' , ID)`
Reads the resource `'type'` with the ID `ID` from the resource file `filename`, and returns a string.

`$$Shell("stringExpr ")` Current value of the exported shell variable {stringExpr }. Note that the braces must be omitted, and the double quotation marks must be present.

`$$Time` Current time. It is useful for time-stamping the resource file. The format is: "hh:mm:ss".

`$$Version` Version number of the resource compiler. ("V1.0")

These variables and functions have numeric values:

`$$Attributes` Attributes of resource from the current resource.

`$$BitField(label , startingPosition , numberOfBits)`
Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

`$$Byte(label)` Returns the byte found at *label*.

`$$CountOf (arrayName)` Returns the number of elements in the array *arrayName*.

`$$Day` Current day (range 1–31).

`$$Hour` Current hour (range 0–23).

`$$ID` ID of resource from the current resource.

`$$Long(label)` Returns the long word found at *label*.

`$$Minute` Current minute (range 0–59).

`$$Month` Current month (range 1–12).

`$$OptionalCount (OptionalName)`
Returns the number of items explicitly specified in the block *OptionalName*.

`$$PackedSize(Start , RowBytes , RowCount)`
Given an offset (*Start*) into the current resource and two integers, *RowBytes* and *RowCount*, this function calls the toolbox routine `UnpackBytes` *RowCount* times. `$$PackedSize()` returns the unpacked size of the data found at *Start*. Use this function only for decompiling resource files. An example of this function is found in `Pict.rez`.

`$$ResourceSize` Current size of resource in bytes. When decompiling, `$$ResourceSize` is the actual size of the resource being decompiled. When compiling, `$$ResourceSize` returns the

	number of bytes that have been compiled so far for the current resource.
\$\$Second	Current second (range 0–59).
\$\$Type	Type of resource from the current resource.
\$\$Weekday	Current day of the week (range 1–7, that is, Sunday–Saturday).
\$\$Word(<i>label</i>)	Returns the word found at <i>label</i> .
\$\$Year	Current year.

Strings

There are two basic types of strings:

Text string	"a..."	The string can contain any printable character except ' ' and '\'. These and other characters can be created through escape sequences. (See Table 15-4.) The string "" is a valid string of length 0.
Hexadecimal string	\$"hh..."	Spaces and tabs inside a hexadecimal string are ignored. There must be an even number of hexadecimal digits. The string \$"" is a valid hexadecimal string of length 0.

Any two strings (hexadecimal or text) will be concatenated if they are placed next to each other with only white space in between. (In this case, returns and comments are considered white space.) Figure 15.4 shows a p-string declared as

```
pstring [10];
```

whose data definition is

```
"Hello"
```

\$05	H	e	l	l	o	\$00	\$00	\$00	\$00	\$00
------	---	---	---	---	---	------	------	------	------	------

Figure 15.4: Internal Representation of a P-string

In the input file, string data is surrounded by double quotation marks ("). You can continue a string on the next line. A separating token (for example, a comma) or brace signifies the end of the string data. A side effect of string continuation is that a sequence of two quotation marks (") is simply ignored. For example,

```
"Hello " "out "
"there."
```

is the same string as

```
"Hello out there.";
```

To place a quotation mark character within a string, precede the quotation mark with a backslash, like this:

```
\"
```

Escape Characters

The backslash character (\) is provided as an escape character to allow you to insert nonprintable characters in a string. For example, to include a newline character in a string, use the escape sequence

```
\n
```

Valid escape sequences are shown in Table 15.4.

<u>Escape Sequence</u>	<u>Name</u>	<u>Hexadecimal Value</u>	<u>Printable Equivalent</u>
\t	Tab	\$09	None
\b	Backspace	\$08	None
\r	Return	\$0A	None
\n	Newline	\$0D	None
\f	Form feed	\$0C	None
\v	Vertical tab	\$0B	None
\?	Rub out	\$7F	None
\\	Backslash	\$5C	\
\'	Single quotation mark	\$27	'
\"	Double quotation mark	\$22	"

Table 15.4: Resource Compiler Escape Sequences

Note to C programmers: The escape sequence \n produces an ASCII code of 13 in the output stream, while the \r sequence produces an ASCII code of 10. This is backwards from the way the C language uses these two characters, so if you are creating string resources that will be used with stdio functions from the standard C library, be sure and use \r in your resource file any time you would use \n in C, and use \n in your resource file any time you would use \r in C.

You can also use octal escape sequences, hexadecimal escape sequences, decimal escape sequences and binary escape sequences to specify characters that do not have predefined escape equivalents. The forms are:

<u>Base</u>	<u>Number</u> <u>Form</u>	<u>Digits</u>	<u>Example</u>
2	\0Bbbbbbbbb	8	\0B01000001
8	\ooo	3	\101
10	\0Dddd	3	\0D065
16	\0Xhh	2	\0X41
16	\\$hh	2	\\$41

Since escape sequences are imbedded in strings, and since these sequences can contain more than one character after the \ character, the number of digits given for each form is an important consideration. You must always code exactly the number of digits shown, using leading zeros if necessary. For example, instead of "\0x4", which only shows a single hexadecimal digit, you must use "0x04". This rule avoids confusion between the numeric escape sequence and any characters that might follow it in the string.

Here are some examples:

```

\077          /* 3 octal digits */
\0xFF         /* '0x' plus 2 hex digits */
\ $F1\ $F2\ $F3 /* '$' plus 2 hex digits */
\0d099        /* '0d' plus 3 decimal digits */

```

You can use the DeRez command-line option `-e` to print characters that would otherwise be escaped (characters preceded by a backslash, for example). Normally, only characters with values between \$20 and \$7E are printed as Apple IIGS characters. With this option, however, all characters (except null, newline, tab, backspace, form-feed, vertical tab, and rub out) will be printed as characters, not as escape sequences.

Using the Resource Compiler

The Resource Compiler is a one-pass compiler; that is, in one pass it resolves preprocessor macros, scans the resource description file, and generates code into a code buffer. It then writes the code to a resource file.

The resource compiler is invoked by the shell's `compile` (or `assemble`) command, just as you would assemble a program. This command checks the language type of the source file (in this case, `rez`) and calls the appropriate compiler or assembler (in this case, the resource compiler). In short, with the exception of a few resource compiler specific options, you use the same commands to create a resource fork from a resource description file that you would use to assemble a program.

Resource Forks and Data Forks

Files on the Apple IIGS actually have two distinct parts, known as the data fork and the resource fork. The data fork is what is traditionally a file on other computers; this is where the executable program is stored, where ASCII text is placed for a text file, and so forth. When the resource compiler writes resources, it writes them to the resource fork of the file. Writing to the resource fork of an existing file does not change the data fork in any way, and writing to the data fork does not change the resource fork. The implications of this can speed up the development cycle for your programs. When you compile a resource description file to create a resource fork for your program, you can and should have the resource compiler save the resource fork to the same file in which the linker places the executable code. When you make a change to your assembly

language source code, you will normally assemble and link the changed program, creating an updated data fork for your program. If the resource description file has not changed, you do not need to recompile the resource description file. The same is true in reverse: if you make a change to the resource description file, you need to recompile it, but you do not need to reassemble or relink your assembly language source file.

Rez Options

The resource compiler supports the `e`, `s`, and `t` flags from the `assemble` or `compile` command. It ignores all other flags.

The resource compiler supports a number of language dependent options. These are coded as the name of the language, an equal sign, and the option list, enclosed in parenthesis. Like the other parameters for the `compile` command, no spaces are allowed outside of the parenthesis.

For example, the following `compile` command uses the options list to specify the `-p` flag, which turns on progress information.

```
compile resources keep=program rez=(-p)
```

The resource compiler will accept up to 31 options in the options list. Any others are ignored.

Here's a complete list of the options that can be used in this options field:

- `-a[ppend]` This option appends the resource compiler's output to the output file's resource fork, rather than replacing the output file's resource fork.

- `-d[efine] macro [=data]` This option defines the macro variable *macro* to have the value *data*. If *data* is omitted, *macro* is set to the null string – note that this still means that *macro* is defined. Using the `-d` option is the same as writing


```
#define macro [ data ]
```

at the beginning of the input.

- `-flag SYSTEM` This option sets the resource file flag for the system.

- `-flag ROM` This option sets the resource file flag for ROM.

- `-i pathname(s)` This option searches the following path names for `#include` files. It can be specified more than once. The paths are searched in the order they appear on the command line. For example,


```
...rez=(-i 13:rinclude:stuff.rez
        -i 13:rinclude:newstuff.rez)
```

- `-m[odification]` Don't change the output file's modification date. If an error occurs, the output file's modification date is set, even if you use this option.

Shell Reference Manual

<code>-ov</code>	This option overrides the protected bit when replacing resources with the <code>-a</code> option.
<code>-p[rogress]</code>	This option writes version and progress information to diagnostic output.
<code>-rd</code>	This option suppresses warning messages if a resource type is redeclared.
<code>-s pathname(s)</code>	This option searches the following path names for resource include files.
<code>-t[ype] typeExpr</code>	This option sets the type of the output load file to <i>filetype</i> . You can specify a hexadecimal number, a decimal number, or a mnemonic for the file type. If the <code>-t</code> option is not specified, the file type of the load file is \$B3.
<code>-u[ndef] macro</code>	<p>This option undefines the macro variable <i>macro</i>. It is the same as writing</p> <pre>#undef macro</pre> <p>at the beginning of the input. It is meaningful to undefine only the preset macro variables.</p>

Note: A space is required between an option and its parameters.

Chapter 16

GSBug Debugger

GSBug is a machine-language debugger that can be used to help find errors in programs. GSBug is available in two different forms: as an application (GSBug) that can be executed from the Finder or from the shell, and as an init (initialization) file (GSBug.init) that is entered automatically when a break point is encountered.

Both versions of the debugger include the following features:

- You can step through your code one instruction at a time (in single-step mode).
- You can step continuously through your code under control of the debugger (in trace mode).
- You can save a trace history to a file on disk.
- You can execute your code, or any portion of your code, at full speed when timing is critical (in real-time mode).
- You can define and insert in your code breakpoints at which the debugger automatically suspends execution.
- You can set a breakpoint so that execution is suspended only after the breakpoint location has been passed a given number of times.
- You can enter the Monitor, execute Monitor commands, and then return to the debugger.
- You can use the debugger's built-in mini-assembler.
- You can view the debugger's Master display, which shows the contents of the Apple IIGS registers, the breakpoints and memory-protection ranges you have set, portions of the stack and memory, and a disassembly of your program's code.
- You can display 368 contiguous bytes of memory starting from the contents of the Direct-Page register.
- You can display memory through templates to show data structures in their proper format.
- You can define and use memory protection windows.
- You can display your program's normal screen in any Apple IIGS display mode.
- You can call up on-line help screens for help with any of the debugger's functions.

The init version is RAM resident and has these features:

- You can view the state of the machine before entry on the Register subdisplay of the Master display.
- You can enter the debugger while your program is running, look at data, set breakpoints, step through code, and resume operation as though nothing has happened.

A CDA called Loader Dumper is also covered in this chapter. The Loader Dumper allows you to interrogate the system loader to find out where the various segments of a program have been loaded. While the Loader Dumper is not technically a part of the debugger, it is most often used in a backup role while debugging a program. In this role, you use the Loader Dumper to locate the segments in the program you are trying to debug.

While all of the information about how to use the debugger is contained in this one chapter, conceptually the chapter has four distinct parts. The first part of the chapter is an overview of the debugger from a conceptual standpoint, describing how to install it and what the principal features are. The second part is a feature-oriented look at the debugger; this second section covers how you would actually use the debugger to load and debug one of your programs. This second part starts with the section "Part 2: Using GSBug. The third part of the chapter, starting with the section "Part 3: GSBug Subdisplay and Command Reference", is a reference manual for the debugger. Finally, "Part 4: Loader Dumper" covers the Loader Dumper CDA.

To get familiar with the debugger, read the first and second parts of the chapter, working through each section in order. To get specific, detailed information about the debugger and its features, refer to Part 3.

Part 1: Getting Started

This section describes restrictions on the use of the debugger and tells you how to get up and running with both versions of the debugger.

The init version of the program is memory resident, so you must move it to your startup disk to load it in RAM. Once you've installed the init version, however, you can run a program to be tested from any disk. With the init version, you can enter debugger while your program is running, look at data, set break points, step through code, and resume work as though nothing has happened.

The application version does not require any installation. You launch the application version the same way you do any other Macintosh application. If you are using the application version of GSBug, you can move the program you want to test on the same disk as GSBug (if the program is small enough to fit) and then load the program into the debugger. You can run the application version under either the Finder™ or the ORCA shell.

Debugger Restrictions

GSBug requires approximately 42000 bytes (\$A400) of memory (not including the memory requirements of the program you are testing). Because the debugger is loaded into memory by the Apple IIGS System Loader and Memory Manager, you have no control over where in memory the debugger is loaded. If your program can be relocated, the System Loader and Memory Manager will load it so that conflicts between the debugger and your program are extremely unlikely. If you write absolute code, however, you will not be able to use it with the debugger if any of the following conditions occur:

- The application writes to the area of memory in which the debugger is loaded.
- The application assigns its direct page or stack into the debugger's code space.
- The application uses the same stack space as the debugger.

In single-step and trace modes, if the application writes or steps to a location between 20 bytes before the beginning of the debugger's stack and 8 bytes after the end of the debugger's stack, the debugger stops executing the program and prints S= on the command line. To continue operation, you must change the value of the S register so that it is outside GSBug's stack range. In real-time mode, a stack conflict can crash the debugger.

Installing the Init Version on Your Startup Disk

This section describes the steps you must take to install the init version of GSBug into memory. It then explains what happens during the startup sequence and how to prevent the debugger from being installed in memory.

If you want the debugger to be installed in memory automatically at startup, you must move it from the Extras disk and place it in the SYSTEM.SETUP folder in the System folder of your startup disk. The init version is a permanent init file called GSBug.init (file type \$B6). You must place it inside the SYSTEM.SETUP folder in the System folder of the startup disk in order for the System to find it at startup.


There are two ways to install the Init version. If you are installing GSBug on a hard disk, the easiest way to install the program is to use the Installer, located on the :ORCA.Extras disk. If you prefer, you can copy the debugger using any copy program you are familiar with; the GSBug Init file is on the extras disk, at path :ORCA.Extras:System:System.Setup:GSBug.

If you are using floppy disks, you can install the Init version in your boot disk, but you will need to delete some files to make room for the program, first. Starting with a copy of the system disk that came with ORCA/M, examine the system folder for files you will not be using. We suggest looking closely at the Fonts folder; the ORCA development environment does not use any fonts. You can also delete all of the CDevs you don't intend to use, and you can even delete any tools you won't be using for your own programs and that are not used by the Finder. (We recommend that only experienced programmers who are very familiar with the tools and when and how they are used attempt to remove any tool files.) The text-based ORCA development that comes with this version of ORCA/M does not use any RAM based tools, although some of the samples do. For samples, see the source code to see what tools are being started. The graphics development environment for ORCA languages, PRIZM, uses all tools required to support New Desk Accessories, plus the Print Manager.

If you no longer want the init version to be installed automatically at startup, remove it from the SYSTEM.SETUP folder of your startup disk. GS/OS looks for the GSBUG.INIT file in the SYSTEM.STARTUP folder at startup; if it finds it there, it automatically installs the file in memory.

To prevent the init version of the debugger from being installed in memory, press and hold down the Option key while booting the System.

During startup, GS/OS allocates memory for GSBug to:


1. patch the CDA vector so that the debugger is able to recognize Control--Option-Esc.
2. patch the break vector to trap breaks in the debugger.
3. install a heartbeat task to ensure that the CDA vector remains patched.
4. enable vertical blanking (VBL) interrupts.

You can unload the debugger at any time with the Unload command described in "Quitting the Debugger" later in this chapter. Unload removes the debugger from memory and restores all the vectors.


The Default Configuration File

The debugger loads a default configuration file called GSBUG.SETUP in the SYSTEM.STARTUP folder of the disk containing the debugger at the initial installation on startup. This file is a standard configuration file that was saved with the CSave command.

Entering the Init Version

This section describes the two ways to enter the init version of the debugger: when you press the Control--Option-Esc key combination while your program is running and when GSBUG takes over the machine through the break vector whenever a break occurs.

While Your Program is Running


To enter the debugger while your program is running (as long as interrupts are not disabled), press Control--Option-Esc. Two possible situations arise:

- If the Q command was used to exit the debugger, the current state of the machine is saved so that it can be restored on exit. In doing so, GSBUG requires a block of memory from the Memory Manager to save the state of the text display.
- If the QR command was used to exit the debugger, the current state of the machine is not restored on exit. This allows you to save your display configurations for use in future sessions.

The debugger also requests a 1K block in bank zero from the Memory Manager. The first two pages of this block are used for the debugger's direct page and stack, and the other two pages are available to the user as a direct page and stack.

If the debugger is called while a ProDOS 8 application is running, no memory is available in bank zero, so the debugger takes a 1K block and restores it on exit.

The Master display appears with the debugger's version number and copyright message across the command line at the bottom of the screen. Both the version number and copyright message disappear as soon as you type anything on the command line.

All the registers, including the stack pointer and program counter, reflect the state of the machine at the moment before you pressed Control--Option-Esc. This means that the stack pointer will not reflect the fact that the return address and processor status were pushed onto the stack when the interrupt occurred.

When a Break Occurs

Whenever a break occurs, GSBUG takes over the machine. It saves the state of the machine and allocates a 1K block. The debugger beeps to let you know that a break has occurred and displays its version number on entry.

All the registers, including the stack pointer and program counter will reflect the state of the machine immediately before the break instruction was executed. The program counter will point

to the break instruction, and the stack pointer will not reflect the fact that the return address and processor status were pushed onto the stack when the instruction was executed.

To instruct the debugger not to trap breaks, you set monitor breaks instead of debugger breaks. See "Using Breakpoints" later in this chapter for details on how to do this.

Entering the Application Version by Launch

You enter the application version of the debugger by launching it the same way you do any other application. Each time you launch the debugger, a default configuration file called GSBUG.SETUP is loaded. It is the standard configuration file that you save with CSave. See "Configuring the Debugger" later in this chapter for details.

If you are launching the debugger from floppy disks, you will find it on the Extras disk at path :ORCA.Extras:Utilities:GSBug. If you have used the installer to install the application version on your hard disk, GSBug will be located in the utilities folder. The utilities folder is located in the same place that you find ORCA.Sys16.

At launch, the debugger asks the Memory Manager for a 1K block in bank zero. The Memory Manager allocates the first two pages of this block for the debugger's direct page and stack; it gives you the other two pages as a direct page and a stack.

The Master display appears with the debugger's version number and copyright message across the command line at the bottom of the screen. These items disappear as soon as you type anything.

Entering the Application Version from the Shell

Like any application, you can run GSBug directly from the shell. From floppy disks, you will need to make sure the disk containing the debugger is installed, then either type the full path name of the debugger, or change the current prefix to the debugger's prefix and type GSBug. The debugger is on the Extras disk at path :ORCA.Extras:Utilities:GSBug.

If you have installed the application version of GSBug on your hard disk, it is installed as a utility. As with any utility, you can execute GSBug from any directory by typing the name of the program, GSBug.

The file type for GSBug is set to S16, so the shell shuts down when you execute GSBug. You can, however, change the file type to EXE using the FILETYPE command. Once the file type is changed, the shell no longer shuts down before launching GSBug. This speeds up the process of launching and returning from GSBug, and it also preserves the state of the shell. It does, however, take up more memory, since both GSBug and the shell remain in memory, so you might want to use the S16 file type in low memory situations.

Loading Your Program into the Application Version

Once you've launched GSBug, the copyright and version number appear on the command line at the bottom of the screen (see Figure 16.1). Then you're ready to load the file to be tested into the debugger. Type the following command to load the file *pathname* into memory:

```
Load pathname
```

Here *pathname* represents the full or partial path name of the program you wish to debug.

Enter the following command to close down an application (execute a System Loader UserShutDown call):

Shutdown *UserID quit_flag*

If any GS/OS (or ProDOS 16) errors are generated during program load or shutdown, the error number appears on the command line. After a successful load, the registers are set as indicated in Table 16.1.

<u>Register</u>	<u>Setting</u>
K/PC	The program bank register (K) and program counter (PC) are set to the starting address of the first segment of the program.
A	The accumulator is set to the user ID of the program loaded. The user ID is assigned by the user ID manager, as described in the Apple IIGS Toolbox Reference.
X, Y	The X and Y index registers are set to 0.
P	The processor status register is set to 0.
D	The direct-page register is set to the bottom of the direct-page/stack space of the program.
S	The stack register is set to the top of the direct-page/stack space.

Table 16.1: Register settings after a successful load

Press Esc to clear the command line.

You can load a maximum of 15 files. If you attempt to load more than 15 files, the error message \$FF appears on the command line.

When you load your program, be sure to make a note of the settings of the K/PC and other registers (in the Register subdisplay at the top of the screen) before you do anything else. After you have used the debugger to run your program, or have reset any registers with debugger commands, you must know the starting location of your program in memory and starting register values in order to run your program again.

Unloading Your Program from the Application Version

The ShutDown call closes down your application. When you make this call, the System Loader responds in one of three ways depending on the value of the quit flag:

1. If the quit flag is 0, the Memory Manager disposes of all memory blocks with the specified user ID and of the ID itself, completely unloading the application.
2. If the quit flag is \$8000, the Memory Manager purges all memory blocks with the specified user ID, making the application quickly reloadable.
3. If the quit flag is any value other than 0 or \$8000, the Memory Manager disposes of all blocks corresponding to dynamic segments with the specified user ID, makes purgeable all blocks corresponding to static segments with the specified user ID, and purges all other blocks with the specified user ID.

In addition, the System Loader deletes all entries for the specified user ID from the Jump Table directory. The application is ready to be revived quickly by the System Loader until any of its static segments are purged.

For details on the UserShutDown call, see the GS/OS Reference.

Setting the Prefix

To change the current GS/OS file prefix , use the following command:

```
Prefix [num] path
```

The *num* field is optional; no entry within the brackets denotes the default prefix.

Displaying the Version Number

Once you've typed something, the version number on the command line of the debugger disappears. To display the debugger's version number and copyright message, type V.

Configuring the Debugger

This section describes three commands and options that configure the Master display of the debugger and save the current configuration to the file of your choice.

- The Set command lets you set up the configuration of the debugger.
- The CSave command allows you to save the current configuration to the file you specify like this:

```
CSave pathname
```

- The CLoad command lets you load the configuration from the file that you specify like this:

```
CLoad pathname
```

When you use the Set command, the debugger enters configuration mode. In this mode, highlighted bars indicate the position of the current stack location (also called the top of the stack) within the Stack subdisplay and the current instruction within the Disassembly subdisplay. In configuration mode the default printer slot is also shown on the command line. Table 16.2 gives details on the keys you press to change the configuration.

<u>Press</u>	<u>Action</u>
left-arrow	Moves the position of the current stack location up one line.
right-arrow	Moves the position of the current stack location down one line.
up-arrow	Moves the position of the current instruction up one line.
down-arrow	Moves the position of the current instruction down one line.
number	Sets default printer slot to number (must be between 0 and 7.)
S	Toggles between absolute and relative stack subdisplay addresses.
D	Toggles memory dump alignment on and off.
Esc	Returns to the command line.

Table 16.2: Keystrokes for Configuring the Debugger

When you use the CSave command, the following information is preserved:

- the position of the current instruction within the Disassembly subdisplay.
- the position of the current stack location within the Stack subdisplay.
- the number of memory-protection ranges and their settings.
- the number of breakpoints and their settings.
- the settings of the RAM subdisplay.
- the stack address format and dump alignment settings.

The debugger loads a configuration file at startup time containing the information listed above. This standard configuration file, GSBUG.SETUP, should be located in the System.Setup folder of the disk containing the debugger; otherwise, it will not load automatically (you can still load it manually).

Debugger Display Screens

When you enter either version of GSBUG, a display similar to the one shown in Figure 16.1 appears on the screen. This display, called the Master display, contains a command line, plus several subdisplays that contain the following types of information:

- the contents of the 65816's registers (the Register subdisplay).
- the contents of the stack (the Stack subdisplay).
- a disassembly of your program's object code (the Disassembly subdisplay).
- the contents of a portion of RAM (memory) that you specify (the RAM subdisplay).
- the breakpoints you have set (the Breakpoint subdisplay).
- the memory ranges you have protected (the Memory-Protection subdisplay).

You also have the option of switching to a display of the contents of the direct page, to a display of the contents of any region of memory you choose, to on-line help, or to any of the display screens normally used by your program.

The next section discusses the commands you can use to select a display. Part 3 of this chapter describes them in detail.

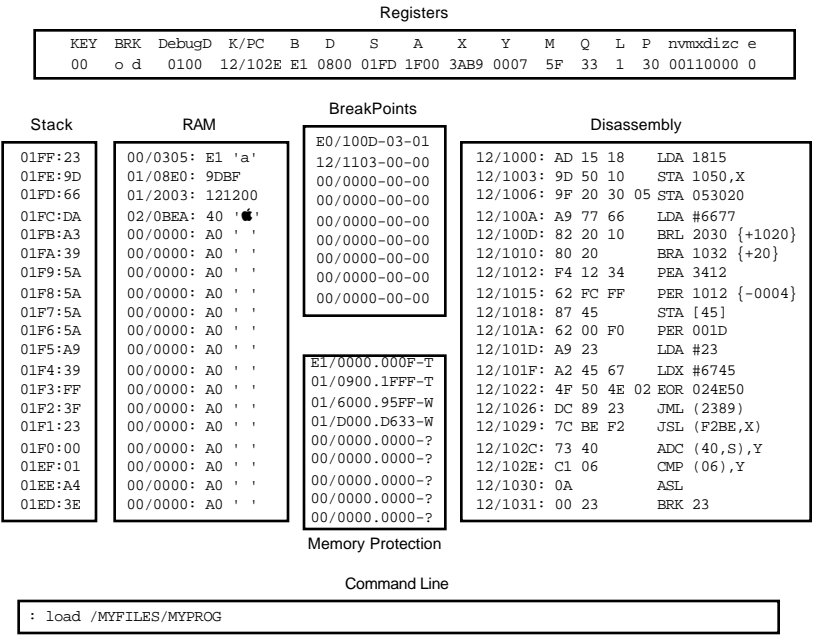


Figure 16.1: Main Display

Switching Displays

When you start GSBug, the Master display appears on the screen. To turn the Master display screen off, type off. To turn the Master display screen on again, type on.

When you turn the Master display off, the standard 80-column screen that is used by Apple IIGS applications appears in its place. You can still type in commands while the Master display is turned off. Any command you enter that uses the Master display automatically turns it on again.

Selecting Displays

Use the commands in Table 16.3 to call other displays.

<u>Display</u>	<u>How to Select</u>
Help Screen	From any display, type a question mark (?) and press Return.
Memory	From the Master display, type the starting address of the memory block you wish to display, followed by a colon (:), and press Return.
Direct-Page Application	From the Master display, type DP: and press Return. To see the display generated by your application, type off on the Master display command line, press Return, and then start your application as described in the section "Running Your Program" later in this chapter. To change the display mode, press one of the keys listed in Table 16.4.
Monitor	To call the Monitor, type mon on the Master display command line and press Return.
Master Display	In Direct-Page or Memory display, press Esc. If your application is being displayed, type on and press Return. From the Monitor, press Control-Y and Return to return to the Master display.

Table 16.3: Selecting Debugger Displays

If the command filter is in effect, you must hold down one or more keystroke-modifier keys to pass commands on to the debugger while your program is running. See the section "The Command Filter", later in this chapter, for more information on this function.

Table 16.4 lists the single keypress commands that work while in single-step or trace modes only; see the section "Single-Step and Trace Modes", later in this chapter, for more information on these keypress commands.

<u>Press</u>	<u>For This Display</u>
1	Text or graphics page 1
2	Text or graphics page 2
4	40-column screen
8	80-column screen
T	Text mode
F	Full-Screen graphics
M	Mixed text and graphics
L	Low-Resolution graphics
H	High-resolution graphics mode
D	Double high-resolution graphics mode
B	Black-and-White (for double high resolution graphics mode)*
C	Color (for double high-resolution graphics mode)*
S	Super high-resolution graphics mode

Table 16.4: Single Key Press Commands to Change Displays in Single-step or Trace Modes

* These commands work only on a color RGB monitor.

Table 16.5 provides key sequences to switch from a display to the command line if you are not in real-time or trace mode. These are escape sequences, so you can type them in at any place on the command line. They are not displayed and have no effect on the command being typed.

<u>Press</u>	<u>For This Display</u>
\1	Text or graphics page 1
\2	Text or graphics page 2
\4	40-column mode
\8	80-column mode
\T	Text mode
\F	Full-Screen graphics mode
\M	Mixed text and graphics mode
\L	Low-Resolution graphics mode
\H	High-resolution graphics mode
\D	Double high/low-resolution mode
\S	Super high-resolution graphics mode
\B	Black-and-White double high-resolution mode
\C	Color double high-resolution mode

Table 16.5: Command-Line Sequences to Switch Screen Displays

The Master Display

In either version of GSBug, the Master display includes information on many aspects of the debugging process. When you start GSBug, a display similar to the one in Figure 16.1 appears. The exact contents of this display depend on the actual contents of memory and on the way in which you have configured the debugger.

This section briefly describes the use of each subdisplay. For detailed descriptions and more in-depth information, refer to the detailed reference sections at the end of this chapter.

The Register subdisplay shows the contents of several 65816 hardware registers, the M and Q pseudoregisters, and some flags and addresses used by the debugger.

The Stack subdisplay shows the contents of 19 bytes of your program's stack. The default location for the stack pointer is the bottom line in the Stack subdisplay, but you can set it to any line you choose with the Set command.

The Disassembly subdisplay shows a disassembly of the machine code in memory using standard assembler mnemonics and address-mode syntax. Disassembly operand formats are shown in Table 16.18. When you start the debugger, this subdisplay is blank. You can assemble a single instruction and display it at the bottom of this subdisplay, or you can disassemble any 19 contiguous lines of code and list them in this field. When you enter trace or single-step modes, a running disassembly of your program appears in the Disassembly subdisplay, with the current instruction highlighted. You can use the Set command to change the line used for the current instruction.

The RAM subdisplay shows the contents of any 19 memory locations you select. You can display each section as a single hexadecimal byte with the equivalent ASCII character (or MouseText character if the high bit is set), as a 2-byte value, or as a 3-byte value. The debugger can also do up to three levels of indirection.

The Breakpoint subdisplay shows from 0 to 17 breakpoint locations you have set. A breakpoint is a point in your code at which you want the debugger to suspend execution so you can examine the contents of memory and the registers. Each breakpoint includes a trigger value—that is, the number of times you want the code at that location to be executed before execution is interrupted, and a trigger count, the number of times the program has actually passed

through this breakpoint so far. You can increase or decrease the number of lines in the Breakpoint subdisplay by simultaneously adjusting the number of lines in the Memory-Protection subdisplay.

The Memory-Protection subdisplay shows memory-address ranges that you have set either to be executed in real time (code-trace ranges, indicated by a T) or to be the only ranges within which code can be executed at all (code-window ranges, indicated by a W). You can increase or decrease the number of lines in the Memory-Protection subdisplay by simultaneously adjusting the number of lines in the Breakpoint subdisplay. You can also set a single line as a trace-history window indicated by an H.

The command-input line (or command line) is used for executing most debugger commands. The only commands not executed from this line are the single-keystroke commands used to control code trace; the cursor movement commands used to enter data into the Master display and the Esc key, which is used to return to the Master display from other displays. See the section "Command-Line Commands" in Chapter 3 for a list of the commands available from the Master display.

Test-Program Display

To turn off the debugger's Master display and show the normal screen display of your program, type off on the command line and press Return. Although GSBug uses only the 80-column text mode, it remembers the last display mode your program was in and switches to that mode when you turn off the Master display.

To change to a different display mode, turn off the Master display, enter trace or single-step mode, and use one of the display-mode commands listed in the section "Single-Step and Trace Modes". To return to the Master display, type on and press Return.

If the command filter is in effect, you must hold down one or more keystroke-modifier keys to pass commands on to the debugger. See the section "The Command Filter" for more information on this function.

Memory Display

You can select a display of the contents of any 336 contiguous bytes of RAM in a single memory bank. To get a Memory display, type the starting address of the memory block, followed by a colon (:), in the command line in the Master display and press Return. For example, to obtain a display of the contents of the 368 bytes starting at address 1100 in bank 12, type the following (the slash (/) is optional):

```
12/1100:
```

You can use the commands listed in Table 16.6 to display the contents of memory on the screen.

<u>Command</u>	<u>Action</u>
<i>address</i> and Return	Displays 16 bytes of memory on command line.
<i>address</i> :	Displays memory starting at <i>address</i> .
<i>address</i> ::	Displays memory starting at the 2-byte address stored at <i>address</i> .
<i>address</i> :::	Displays memory starting at the 3-byte address stored at <i>address</i> .
<i>address1.address2</i> :	Displays memory from <i>address1</i> to <i>address2</i> .
DP:	Displays the contents of the Direct page.

Table 16.6: Displaying memory on the screen

If you request more bytes than the debugger can display, only the bytes that can fit on the screen appear. To display the next screen full of memory, press the Space bar to display the next address on the command line, and then press Return.

A sample Memory display is shown in Figure 16.2. Each line begins with the memory address of the first byte shown on that line, followed by the contents of 16 memory locations. The memory contents are shown first as hexadecimal values and then as their equivalent ASCII characters. Table 16.7 shows how the character set is displayed.

<u>ASCII Value</u>	<u>Displayed As</u>
\$00–\$1F	.(period)
\$20–\$7F	Normal video
\$80–\$9F	.(inverse-video period)
\$A0–\$FF	Inverse video

Table 16.7: Character set for memory contents

```

12/1100: 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 .....
12/1110: 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 .....
12/1120: 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 !"#$%&'()*+,-./0
12/1130: 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 123456789:;<=>?@
12/1140: 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 ABCDEFGHIJKLMNOP
12/1150: 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 QRSTUVWXYZ[\]^_`
12/1160: 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 abcdefghijklmnop
12/1170: 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 qrstuvwxyz{|}~■
12/1180: 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 .....
12/1190: 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 .....
12/11A0: A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 !"#$%&'()*+,-./0
12/11B0: B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 123456789:;<=>?@
12/11C0: C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 ABCDEFGHIJKLMNOP
12/11D0: D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 QRSTUVWXYZ[\]^_`
12/11E0: E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 abcdefghijklmnop
12/11F0: F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF 00 qrstuvwxyz{|}~■
12/1200: 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 .....
12/1210: 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 .....
12/1220: 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 !"#$%&'()*+,-./0
12/1230: 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 123456789:;<=>?@
12/1240: 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 ABCDEFGHIJKLMNOP
12/1250: 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 QRSTUVWXYZ[\]^_`
12/1260: 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 abcdefghijklmnop
:12/1270:

```

Figure 16.2: Sample Memory Display

Press Esc or type on to return to the Master display.

If you enter any command other than one that displays or sets memory, the Master display replaces the Memory display and the command is executed.

An ASCII equivalent of memory is displayed to the right of the dump.

The Memory display is aligned initially. When the display is aligned, the byte stored at XX/XXXn is displayed in the nth position on the screen. If the display is not aligned, the first byte in the dump appears in the first position.

Use the Set command to change the Memory display alignment. See "Configuring the Debugger" earlier in this chapter for details.

To change the contents of memory, use the commands described in the section "Altering the Contents of Memory" later in this chapter. To return to the Master display, press Esc.

Displaying Memory on the Command Line

You can display 16 bytes of memory on the command line with the Quick-Dump command. Simply type the starting address for the 16-byte display and press Return.

For example, you might type 12 and press Return for the following command-line display:

```
:01/0012: FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF
```

The Quick Dump display remains on the command line until you type any key.

Direct-Page Display

You can select GSBug's Direct-Page display by typing DP: on the command line and pressing Return. This display shows 368 bytes starting with the address in the D register. The Direct-Page display is identical in appearance and function to the Memory display that you would obtain for a block of memory starting at the address in the D register.

Help Screens

To display a help screen showing the commands available at any time, type a question mark (?). To return from the help screen to the display from which you called it, press any key except Esc. To return to the Master display, press Esc.

From within any of the subdisplays in the Master display, type ? to view a help screen with commands relevant to that subdisplay.

Stepping Through Your Program

You can step through your program one instruction at a time or continuously, with GSBug intercepting and interpreting each instruction. Executing your program in this fashion gives you maximum control over the process, allowing you to stop at any point and examine the contents of the registers or your program's display. For timing-critical programs and sections of programs, you can also execute the code at full speed. See "Running Your Program" later in this chapter for explanations of how to:

- step through your program one instruction at a time (single-step mode).
- step continuously through your program (trace mode).
- execute your program at full speed (real-time mode).
- set and use breakpoints.
- set and use memory-protection ranges.

Printing Debugger Screens

You can print the current 80-column debugger text screen, including help screens and your program's display, by typing P on the command line and pressing Return. This prints the current screen to the printer in the default printer slot, which is initially slot 1. You cannot use the P command to print graphics screens. You can print the current text screen to the printer in a given slot with the command

P num

To change the slot to which the debugger assumes the printer is connected, use the Set command as described in the section "Configuring the Master Display".

GSBug does not print using the .PRINTER driver, so the printer initialization string and other printer variables you set for the .PRINTER driver are not active when you are in the debugger. If you use the Apple IIGS computer's built-in printer port, you can use the Control Panel to set printer communication options.

Using Monitor Routines

The Apple IIGS Monitor consists of a set of ROM-based routines that you can use to perform many functions not otherwise available from the debugger. The Monitor provides the following features, so you can:

- examine the contents of any locations in memory, including ROM routines.
- change the contents of any locations in RAM.
- copy a block of data from one location in memory into another and verify that the contents of the two blocks of memory are identical.
- clear a range of memory.
- search for one or more bytes within a range of memory addresses.
- examine and change the contents of Apple IIGS registers.
- convert hexadecimal numbers to decimal and vice versa.
- perform hexadecimal addition and subtraction.
- run a machine-language program that is in memory.
- enter machine-language programs directly from the keyboard, using standard 65816 mnemonics (this Monitor routine is called the Mini-Assembler).
- disassemble a range of addresses.
- start Applesoft BASIC.
- change the screen display.
- change the cursor symbol.
- redirect input and output links.
- call Apple IIGS tools.
- set and display the system clock time and date.

To enter the Apple IIGS Monitor, type `mon` on the debugger's command line and press Return. To return from the Monitor to the debugger, press Control-Y and press Return. The Apple IIGS Monitor is described in detail in the *Apple IIGS Firmware Reference*.

Note: the registers or stack may be changed while you are in the monitor.

Quitting the Debugger

There are three commands that let you exit the debugger:

- `Q` lets you quit the debugger from either the application or the init version. It restores the state of the machine for the next startup.

Note: The `Q` command only saves the display state and the state of the registers.

Warning: For the init version only: if you've changed memory (or if memory has been changed), the debugger may crash when quitting.

- The `QR` command only applies to the init version of the debugger. `QR` also allows you to exit the debugger but does not restore the state of the machine at startup— rather it is left as modified during the current session. See "Entering the Init Version" earlier in this chapter for details.

- Unload allows you to unload the debugger from memory and then quit (not available in the application version).

Warning: Be sure to remove real breakpoints with the Out command before exiting the debugger. (See "Using Breakpoints" for details on the Out command.) If the breakpoints are not removed, the break instructions stay in memory so that the original instructions cannot be restored.

If you exit the debugger with the Q command, you instruct the debugger to restore all display screens to their state prior to launch and then jump to the address indicated by the current value of the program counter (K/PC). See "65816 Registers" for details. Upon quitting with the Q command, the debugger also deallocates all the memory allocated during that session, including the 1K direct-page and stack blocks, and any files loaded with the Load command. The process of unloading the debugger completely removes it from memory with the following steps:

1. It removes the heartbeat task and restoration of the CDA vector.
2. It checks the break vector to see if it has been altered by the user. If the break vector has not been changed, the debugger restores it to its value at first patch during initialization. If the user has modified the break vector, the debugger does not restore it.
3. It deallocates the memory in which the debugger was loaded.

Part 2: Using GSBug

GSBug allows you to load your program into memory and run through it under the debugger's control. As the program executes, you can examine the contents of the 65816's registers, your program's direct page and stack, and any memory locations in which you are interested. You can interact with the program as required, returning to the debugger's display when the program reaches the breakpoints that you set or when it crashes.

GSBug can display an assembly-language disassembly of your program's machine code. It cannot execute your source code or recreate your source code from machine code. Therefore, the debugger is easiest to use with assembly-language programs. However, even if your program was written in a higher-level language and you have no knowledge of assembly language, you can use the debugger to determine in which load segment the problem lies. You can also gain a better understanding of your program's operation by examining the contents of the stack, direct page, memory, and registers.

This part of the chapter begins with a discussion of how to get help and run your program through the debugger. We do not have the space to examine in detail all of the abilities of GSBug in this topical overview, but the remainder of the chapter provides you with some hints that should help you start debugging your programs.

Getting Help

To display a help screen with a list of important debugger commands, type a question mark (?) on the command line and press Return. To return to the command line of the Master display, type ?.

To display a help screen for any of the subdisplays within the Master display, type a question mark from within the desired subdisplay. To return to the command line of the Master display, press Esc. To return to the subdisplay associated with that help screen, press any other key.

Running Your Program

You can step through your program one instruction at a time or continuously, with GSBug intercepting and interpreting each instruction. Executing your program in this fashion gives you maximum control over the process, allowing you to stop at any point and examine the contents of the registers or your program's display. For timing-critical programs and sections of programs, you can also execute the code at full speed. This section explains how to:

- step through your program one instruction at a time (single-step mode).
- step continuously through your program (trace mode).
- execute your program at full speed (real-time mode).
- set and use breakpoints.
- set and use memory-protection ranges.

In single-step and trace modes, if the application writes or steps to a location between 20 bytes before the beginning of the debugger's stack and 8 bytes after the end of the debugger's stack, the debugger stops executing the program and prints S= on the command line. To continue operation, you must change the value of the S register so that it is outside the debugger's stack range. In real-time mode, a stack conflict can crash the debugger.

Single-Step and Trace Modes

In both single-step and trace modes, the code appears in the Disassembly subdisplay as it is being executed, with the current instruction (the one that is about to be executed) highlighted. In both modes the code scrolls up the screen.

Use the Set command to change the position of the current instruction. See "Configuring the Debugger" earlier in this chapter for details.

In single-step mode you can step through your program one instruction at a time. As each instruction is about to be executed, it is highlighted in the Disassembly subdisplay of the Master display.

In trace mode GSBug automatically steps through each instruction in succession; trace mode is identical to single-step mode except that it is free-running.

During code execution, the lines above the current instruction display the instructions that have been executed. The lines below show the instructions following the current instruction. Note that these lines are not necessarily the same as the instructions that will be executed. For example, if a jump instruction follows the current instruction, the instruction after the jump will probably not be executed but only displayed.

The current state of the m, x, and e bits determines the operand lengths of the instructions displayed beneath the current instruction. The Disassembly mode flag (d) has no effect during a trace or single-step.

Real breakpoints appear as BRK instructions below the current instruction; however, when the debugger actually reaches the real breakpoints, the actual instruction – not the BRK instruction – is shown.

Use the command-line commands in Table 16.8 to initiate single-step and trace modes. The command line is still active if you turn the Master display off to see your program's display.

<u>Command</u>	<u>Action</u>
S	Enters single-step mode at the current instruction. The current instruction is the next instruction to be executed as indicated by the K/PC register. The K/PC register is updated by the debugger each time an instruction is executed in single-step or trace modes, each time a new program is loaded, and each time you execute a K=, PC=, or K/PC= command. The current instruction appears at the highlighted line of the Disassembly subdisplay; press the Space bar to execute it, or press Return to enter trace mode.
<i>address</i> S	Enters single-step mode at <i>address</i> . The K/PC register is set to <i>address</i> and the instruction at address appears at the highlighted line of the Disassembly subdisplay; press the Space bar to execute it, or press Return to enter trace mode.
T	Enter trace mode at the current instruction (as indicated by the K/PC register and the highlighted line of the Disassembly subdisplay). The debugger begins executing code immediately and continues to execute instructions until you press Esc to stop it or until it reaches a breakpoint or a BRK instruction.
<i>address</i> T	Enters trace mode at <i>address</i> . The K/PC register is set to <i>address</i> , and the debugger begins executing code immediately. The debugger continues to execute instructions until you press Esc to stop it, or until it reaches a breakpoint or a BRK instruction.

Table 16.8: Commands for Initiating Single-Step and Trace Modes

When you load your program, be sure to make a note of the values of all the fields in the Register subdisplay before you do anything else. After you have used the debugger to run your program, or have reset any registers with a debugger command, you must know the starting location and register settings for your program in order to run it.

Once you are in either single-step or trace mode, you can use any of the keypress commands in Table 16.9. The commands that change display modes are intended for use with your program's display. Do not use them when the debugger's Master display is on the screen.

<u>Command</u>	<u>Action</u>
Esc	Terminates trace or single-step mode and return to the command line.
Space bar	Single-steps one instruction (or enters single-step mode if in trace mode).
Return	Starts continuous tracing.
R	Traces until the next RTS, RTI, or RTL. This command allows you to trace through one subroutine at a time.
J	Begins to execute code in real time at the current instruction.
X	If the current instruction (the next to be executed) is a JSL or JSR, executes in real time until the matching RTL or RTS. If the next instruction is not a JSL or JSR, ignores this command.
down-arrow	Skips the next instruction. You can use this command to skip a BRK instruction, for example.
Q	Toggles the sound on or off. If the sound is on, the speaker beeps each time an instruction is executed.

1	Changes the display to text or graphics page 1. Use this command when in 40-column text mode or mixed text and graphics mode.
2	Changes the display to text or graphics page 2. Use this command when in 40-column text mode or mixed text and graphics mode.
4	Changes the display to a 40-column screen. Use this command when in text mode.
8	Changes the display to an 80-column screen. Use this command when in text mode.
T	Changes the display to text mode.
F	Changes the display to full-screen graphics mode.
M	Changes the display to mixed text and graphics mode.
L	Changes the display to low-resolution graphics mode.
H	Change the display to high-resolution graphics mode.
D	Changes the display to double high/low resolution graphics mode.
S	Changes the display to super high-resolution graphics mode. This is the normal Apple IIGS display mode.
B	Changes the display to black-and-white, double high-resolution graphics mode.
C	Changes the display to color, double high-resolution graphics mode.
left-arrow	Changes to the slow trace rate.
right-arrow	Changes to the fast trace rate.
⌘	Pauses the trace until the Apple key is released.
?	Displays the command help screen.

Table 16.9: Single-Step or Trace Mode Commands

If the command filter is in effect, you must hold down one or more keystroke-modifier keys to pass commands on to the debugger. See "The Command Filter" later in this chapter for more information on this function.

Both trace and single-step modes are terminated under three conditions:

- the occurrence of a BRK instruction.
- the execution of code that is not in any code-window range.
- the encounter of a JSL in a code-trace range.

If a breakpoint is triggered while the debugger is in trace mode, execution immediately terminates. If a breakpoint is triggered in single-step mode, the debugger will beep, but it will remain in single-step mode. Both tracing and single-stepping will also terminate immediately if code is executed that is not in any code-window range.

In addition, if the debugger encounters a JSL in a code-trace range during trace or single-step mode, the subroutine is automatically run in real-time. See "Using Memory Protection Ranges" later in this chapter for details.

The debugger recognizes two GS/OS entry points: the standard GS/OS entry point at \$E100A8 and the stack-based entry point at \$E100B0. If the debugger encounters a JSL to either address, it automatically executes the call in real-time.

Trace or single-step modes still operate if the display is turned off with the Off command. When code execution terminates for any reason, the display turns back on automatically, and the Disassembly subdisplay looks the same. Tracing with the display off is much faster than with it on.

Saving and Viewing a Trace History File

You can save a history of a trace to a file on a disk. A trace history saves all the instructions inside a history window while you are tracing or single-stepping through a program. A trace history file also saves the state of all registers when the debugger executed an instruction.

Table 16.10 lists the commands for saving and viewing a trace history.

<u>Command</u>	<u>Action</u>
OpenHist <i>pathname</i>	Opens the file specified by <i>pathname</i> as a history file.
CloseHist	Closes the currently open history file.
ViewHist <i>pathname</i>	Displays the history file specified by <i>pathname</i> in the Disassembly subdisplay.

Table 16.10: Trace History Commands

When you open a history file, the debugger preserves all instructions executed within the history window to the specified file, including instructions from several separate traces. Make sure you close your history file when you have completed the execution of the desired code.

If an already existing history file remains open, GSBug will overwrite it.

Attempting to open a history file when there is a currently open one is an error. If a disk error occurs while the debugger is recording a trace history, the history file is closed automatically.

You can set a single line as a trace history window in the Memory-Protection subdisplay with the H command. See the "Setting History Windows" subsection in "Memory Protection" later in this chapter. If no trace history windows are set, GSBug saves all instructions that are executed.

Use the ViewHist command to view a saved history file in the Disassembly subdisplay. The saved history display:

- highlights the current instruction.
- shows the current history step on the command line.
- sets the registers on the top of the screen to values held just before execution of the current instruction.

Although the Stack register changes during a trace history, the Stack subdisplay does not change because the debugger does not save any memory locations with the trace history. Thus the values on the stack when you are viewing the history are not the same as the values present at program execution.

Table 16.11 describes the commands you can use to control the viewing of the history.

<u>Command</u>	<u>Action</u>
up-arrow	Goes to previous step.
down-arrow	Goes to next step.
⤴-up-arrow	Goes up one page (19 steps).
⤴-down-arrow	Goes down one page (19 steps).
right-arrow	Goes to last step.
left-arrow	Goes to first step.
<i>number</i> (press Return)	Goes to step specified by number.
Esc	Exits history display.

Table 16.11: Commands to control the viewing of history

The trace history records only the instructions within the history windows; therefore, two instructions that look contiguous while you are viewing the history display may not have been contiguous while the program was running.

Be aware that history files can become extremely large; each step of saved history occupies approximately 26 bytes.

Real-Time Mode

In real-time mode the debugger passes control of the computer to the code that you specify. The code runs at full speed, just as it would if you were running it without the debugger.

Before you try running your program in real-time mode, read through the rest of this "Running Your Program" section. As described in the following subsections, you can exercise control over the execution of your program, even if it is running in real-time mode. You can cause execution to stop automatically at any breakpoints you specify (see "Breakpoints"). You can run your entire program in real-time mode, or you can specify that the code in certain memory ranges is to be run in real-time mode while the rest of the program is run in trace or single-step mode.

Use the command-line commands in Table 16.12 to initiate real-time mode.

<u>Command</u>	<u>Action</u>
<i>address</i> x	Executes a JSL directly to code at address. If you omit the address, uses the current K/PC address is used. (Note that if you omit the address, you must use an uppercase X for this command.) This command assumes your routine ends in an RTL; the code is executed in real-time mode and when the RTL is executed control returns to the debugger. The debugger automatically turns off the Master display before executing this command.
x	Forces a JSL to the code at the current K/PC. Routine must end in RTL.
<i>address</i> J	Jumps directly to the code at the address. If you omit the address, the current K/PC address is used. This command executes an unconditional jump to address and the code at that address is executed in real-time mode. Control does not return to the debugger unless you have set a real breakpoint or a non-breakpoint BRK is executed (while breakpoints are set to DBRK). See "Breakpoints" for information on the use of breakpoints.

	The debugger automatically turns off the Master display before executing this command.
J	Jumps to the code at the current K/PC.
<i>address</i> G	Jumps to the code at address. If you omit the address, the current K/PC is used.
G	Jumps to the code at the current K/PC. (identical to the J command described earlier in this table).

Table 16.12: Initiating Real-Time Mode

The debugger terminates real-time execution of code under the following conditions:

- It encounters a break instruction.
- A real breakpoint is triggered by the program.
- The program returns from the JSL (as it does in the X command).

Real breakpoints must be in if they are to work during real-time execution.

Do not use the Control-⌘-Option-Esc keypress to terminate real-time execution of code. Strange and unexpected results may occur.

The Command Filter

GSoft usually intercepts all keystrokes that occur while tracing or single-stepping through code and passes them to the debugger's command interpreter. If the program you are debugging requires input from the keyboard, you can set the debugger to pass all keystrokes on to the application unless one or more keystroke-modifier keys are also pressed. To select the key or keys to be used as the keystroke modifiers, use the Key command as described in "Keystroke Modifiers".

Keystroke modifiers prevent debugger commands from interfering with your program. For example, suppose your program has menu options that are activated by key presses and that Q causes the program to quit. You can set the keystroke modifier so that commands are passed to the debugger only when you hold down the Option key. Then to toggle sound on or off without causing your test program to quit, you would press Option-Q.

Remember that when you have set a keystroke modifier, you must use that key or key combination in order to send any command to the debugger while in trace or single-step mode. For example, to quit trace mode when Option is set as the keystroke modifier, press Option-Esc; to single-step one instruction, press Option-Space bar.

The current value of the keystroke modifier appears in the Register subdisplay. See the subsection "Keystroke Modifiers" in "Debugger Registers" for a list of the bit assignments. To set the keystroke modifier, use the command

key=keynum

Memory Protection

GSoft allows you to specify address ranges within which code is executed at full speed (code-trace ranges) and to specify ranges outside which no code is executed (code-window ranges). Instructions for setting these ranges are given in "Memory-Protection Subdisplay".

Memory-protection ranges work only in trace and single-step modes; they do not function in real-time mode.

All code inside a code-trace range (indicated by a T on the Memory-Protection subdisplay) is executed automatically in real time. When your code executes a JSL to this memory-protection address range, the code inside this range is executed in real time. When the matching RTL is encountered, execution returns to single-step or trace mode.

Use code-trace ranges to specify subroutines that must be executed at full speed, such as disk I/O routines. Note that interpreted breakpoints do not function inside code-trace ranges; you must insert real breakpoints to stop execution within a code-trace range. When you start the debugger, a code-trace range is automatically set for the range E1/0000 through E1/000F so that Apple IIGS tool calls are run at full speed.

If one or more code-window address ranges (indicated by W's on the Memory-Protection subdisplay) are specified, code is executed only if it is inside one of the code-window ranges. Execution stops anytime the program counter (K/PC) equals an address not in any of the code-window address ranges. (If you don't specify any code-window address ranges, code can be executed at any address.)

You can use code-window ranges to protect code outside your program's normal code from being executed. For example, if a bug in your code is causing the system to crash by executing code in the wrong location in memory, you can restrict execution to protect that area of memory by making sure it lies outside any code-window ranges. Then, when your program jumps to that area of memory, execution stops and you can attempt to find the location and nature of the bug.

Setting Trace History Windows

You can set a history window within the Memory Protection subdisplay by typing H while editing a line within the display. Then the debugger will save only those instructions within that line to the history file. (If no windows are set, the debugger saves all the instructions.) See "Saving and Viewing a Trace History File" earlier in this chapter for details.

Breakpoints

A breakpoint is a location at which the debugger suspends execution of the program, giving you the opportunity to examine the disassembly and the state of the machine at that location. GSBug allows you to set up to 17 memory addresses as breakpoints. Breakpoints can be either real or interpreted. A real breakpoint is a BRK instruction that the debugger has inserted in the code. An interpreted breakpoint is a memory address at which the debugger suspends execution in trace mode. When you set interpreted breakpoints, the debugger compares the address of the instruction about to be executed (that is, the program counter) to the breakpoint addresses before executing each instruction.

Instructions for setting breakpoints are given in the section "Breakpoint Subdisplay". To make breakpoints work when the debugger is running in real-time mode, you must insert real breakpoints into the code. To set real breakpoints, set the breakpoint addresses as described in "Breakpoint Subdisplay", press Esc to return to the command line, type `in`, and press Return. The letter `i` is displayed under BRK in the upper left corner of the Master display to indicate that real breakpoints are `in`. To remove real breakpoints, type `out` on the command line and press Return. The letter `o` is displayed under BRK in the upper left corner of the Master display to indicate that real breakpoints are `out`.

You must set a trigger value for each breakpoint. The trigger value specifies the number of times that the instruction at the breakpoint address must be encountered before the debugger suspends execution. Trigger values are 1-byte hexadecimal values. For example, if you set the trigger value for a breakpoint to 2, the debugger executes that instruction the first time it

encounters it but suspends execution the second time. To disable a breakpoint without removing the breakpoint address, set the trigger value to 0; breakpoints with trigger values of 0 are ignored by the debugger.

In trace mode execution stops when the trigger value is reached for both real and interpreted breakpoints. In real-time mode execution stops when the trigger value is reached for real breakpoints only. In single-step mode, the computer beeps when the trigger value is reached.

In trace and single-step modes, execution stops anytime the program encounters a BRK instruction that is not set as a breakpoint. In real-time mode, you can select whether a non-breakpoint BRK causes a return to the debugger or an exit to the Monitor. To have non-breakpoint BRK instructions return you to the debugger, type `dbrk` on the command line and press Return. A `d` appears next to the `i` or `o` in the upper left corner of the Master display. To have non-breakpoint BRK instructions cause an exit to the Monitor, type `mbrk` on the command line and press Return. An `m` appears in the upper left corner of the Master display. The default mode is DBRK; you are returned to the debugger when a non-breakpoint BRK is encountered during real-time execution.

When a breakpoint is triggered, execution stops and you can check the contents of the registers and memory locations shown in the Master display or Memory display. To resume execution in trace mode, type `T` on the command line and press Return. The computer beeps when an interpreted breakpoint is triggered in single-step mode. Continue pressing the Space bar to continue stepping through the code. When a BRK instruction that you did not set as a breakpoint is encountered in trace or single-step modes, execution stops, the computer beeps, and the Register subdisplay contains information about the state of the machine when the break occurred. Use the `On` command to see the Master display if necessary.

To clear all breakpoints (when real breakpoints are out), type `Clr` on the command line and press Return.

Debugging Segmented Programs

In order to use GSbug to debug a segmented program, you must know where each segment has been loaded in memory. In the case of dynamic segments, you must know not only where the segment has been loaded, but whether it has been loaded. This information is available through Loader Dumper, a desk accessory provided with GSbug. Loader Dumper is described later in this chapter.

Use the following procedure to load your program using GSbug and determine where each segment is loaded in memory:

1. From the ORCA shell's command line, type `GSbug` and press Return to call the debugger. The debugger Master display will appear on the screen.
2. Type `Load pathname`, where *pathname* is the path name of the program you want to debug, and press Return. Your program is now loaded into memory.
3. Press `-C`-Control-Esc to get the Desk Accessories menu.
4. Select Loader Dumper from the Desk Accessories menu. The Loader Dumper main menu will appear on the screen. (Note: The Loader Dumper is not preinstalled on our system disk. You must install it before it can be used.)
5. Select Dump Path Name Table from the Loader Dumper main menu. The path name table provides a cross-reference of path names and user IDs.
6. Scroll through the path name table (by pressing Return for each path name) until you find the path name of the program you are testing. Make a note of the user ID of the program.

7. Press Esc to get back to the Loader Dumper main menu.
8. Select Get UserID Information from the main menu.
9. Type the user ID of your program in response to the prompt that appears on the screen.
10. A listing of all the load segments in your program appears on the screen. Write down the memory locations of all the segments.
11. Press Esc to return to the main menu, press Esc again to quit Loader Dumper, and then select Quit from the Desk Accessories menu to return to the debugger.

Several possible courses of action are now open to you. If you have no idea in which load segment your program is crashing, you can start by running the program until it crashes and then examining the debugger display to determine the location of the problem instruction. If you know in which segment the problem lies, you can go immediately to that segment, or you can set a breakpoint at the beginning of that segment and run the program until it stops automatically at that breakpoint.

Watching a Running Disassembly

If your program does not require any input from the keyboard, you can watch a disassembly on the debugger screen as the program executes to find the exact location at which it goes astray. This technique will probably be useful only for short programs or programs that crash almost immediately upon execution, because the program will execute very slowly while the debugger display is on the screen.

Use the following procedure to run your program under control of GSDebug, with a running disassembly appearing on the screen:

1. If you have not already done so, load your program as described in the preceding section. Write down the information in the Register subdisplay of the debugger so that you can return the machine to its initial state each time you run your program.
2. Type `s` and press Return. The debugger is now in single-step mode, starting with the first instruction of your program. Each time you press the Space bar, the instruction highlighted in the Disassembly subdisplay is executed. To return to the debugger's command line, press Esc. Watch the contents of the registers and the stack as you execute commands. If any specific memory locations are critical to the execution of the program, display those locations in the Memory subdisplay of the Master display.

To execute commands automatically in quick succession (that is, to enter trace mode), press Return. To start trace mode from the debugger's command line, type `t` and press Return. Your program begins executing under debugger control, one instruction at a time in rapid succession. The speaker beeps each time an instruction is executed. You can turn off the speaker by pressing `Q`. You can stop execution at any time by pressing Esc.
3. When your program executes a BRK instruction, the disassembly stops scrolling. The last execution executed (the BRK instruction) is highlighted. The last several instructions executed appear above the current instruction. A BRK instruction is actually a null (a 0 byte). Since a BRK instruction is not a normal part of a program, the fact that your program executed one means that some previous instruction (contrary to your intent when you wrote the program) sent the program to the wrong place in memory. With luck, the instruction that did it will still be on the screen.

If the offending instruction is no longer on the screen, set a code-window range as described in the section "Using Memory-Protection Ranges" later in this chapter and run the program again.

Remember to restore all the fields in the debugger's Register subdisplay to their original values before attempting to rerun the program. If you do not, the program will probably not run correctly, through no fault of its own.

Using Breakpoints

If you have to interact with your program for it to run, if you have some idea of which segment contains the bug, or if you just want to execute the program more quickly, you can set one or more breakpoints before running the program. To set breakpoints and run the program under debugger control, use the following procedure:

1. If you have not already done so, load your program. Write down the information in the Register subdisplay of the debugger so that you can return the machine to its initial state each time you run your program.
2. Use the Loader Dumper routine to determine the starting locations of the load segments of your program.
3. Back in the debugger, type `bp` and press Return in order to specify breakpoints. Following the instructions in the section "Breakpoint Subdisplay", set breakpoints at the beginning of each load segment (if you do not know in which segment the bug lies) or at the beginning of any segment that you want to examine more closely.
4. If you must interact with the program for it to run, set a keystroke modifier that will not interfere with your program. (The keystroke modifier is a key that you must press simultaneously with any key that you want the debugger to interpret as a debugger command. For example, if you set the Option key as the keystroke modifier, you must press Option-Esc to terminate trace mode. If you do not press the keystroke-modifier key, the debugger ignores the keypress and your program is free to act on it.)
5. Type `off` and press Return. The GSbug display clears so you can see the normal display of your program. Trace mode also runs more quickly with the debugger's display turned off.
6. Type `T` and press Return. Your program will begin executing under debugger control, one instruction at a time in rapid succession. The speaker beeps each time an instruction is executed. You can turn off the speaker by pressing `Q`. (Remember also to press any keystroke-modifier key you have set.) Interact with your program as you normally do. (It will run more slowly than normal.) You can stop execution at any time by pressing Esc. (Remember the keystroke modifier.)
7. When the debugger comes to a breakpoint, the debugger's Master display appears on the screen. The number shown under K/PC in the Register subdisplay indicates the location of the instruction at which the program stopped. To see a disassembly of the program starting at the breakpoint location, type the K/PC address followed by an `L` (for example, `010240L`) and press Return.
8. If you are zeroing in on the bug, you might want to use single-step mode with the debugger's Master display on the screen. To step through the segment one instruction at a time, type `S` and press Return. Now each time you press the Space bar (remember the keystroke modifier), one instruction executes. You can watch the contents of the stack

and the machine's registers as each instruction executes. You can also display the contents of up to 19 memory locations, which you can watch as the program executes.

9. To return to executing the program from the Master display, press Esc to exit single-step mode, and then type OFF and T as before. Each time the debugger gets to a breakpoint, it will return you to the Master display.

If a dynamic segment is loaded anytime during execution of the program, you can pause execution of your program and go back to Loader Dumper to find out where it has been placed in memory.

Breakpoints can be used for other purposes than finding a particular segment. Suppose, for example, that your program seems to run well at first, then crashes after having lulled you into a false expectation of success. In this case some routine may be failing after going through several iterations. To handle such a situation without stopping the program every time the routine is executed, you can include a trigger value for a breakpoint. The debugger counts the number of times it encounters the breakpoint and suspends execution only when the trigger value is reached.

If you must execute a routine at full speed in order for it to work correctly, you can insert real breakpoints into the code with the In command. When you do so, the debugger actually inserts BRK instructions into memory at the breakpoint locations. Trigger values work for real breakpoints that you have set; the debugger will still suspend execution whenever it encounters a BRK instruction that you did not set as a breakpoint.

Using Memory-Protection Ranges

It may be that certain portions of your code must be executed at the full speed of the 65816 microprocessor, or you may be fairly sure that some parts of your program are working correctly, and you don't want to trace through them. To make this happen automatically every time you trace through the program, you can set any areas of memory you choose as code-trace ranges. When the program executes a jump to a location within a code-trace range, the debugger relinquishes control to your program and the code executes at full speed. The portion of memory used to run tool calls is automatically set as a code-trace range when you load the debugger.

You can also set one or more portions of memory (the limits of your code as revealed by Loader Dumper, perhaps) as code-window ranges. If the program attempts to execute code outside the code-window ranges you have set, execution stops. You might want to set a code-window range, for example, if your program is executing a jump to execution at some incorrect memory location and trashing memory before it stops so that you have to reboot the machine every time you try to run the program with the debugger.

If your program loads a dynamic segment during execution and you want to pause as soon as control is transferred to the dynamic segment, you can set code window ranges to include all the static segments at the start of the program. Then when the dynamic segment is loaded and control is transferred to it, the program will be outside any code window range and execution will stop.

Once you have set any code-window range, no code that is not in a code-window range will be executed. If you set a code window range equal to the memory location of one of your program segments, you must set code-window ranges for all other segments that you want to run. Also remember to set the portion of memory used to run tool calls (E1/0000–E1/000F) to a code-window range if your program makes any tool calls.

Debugging Multi-Language Programs

One of the advantages of the ORCA development environment is that it allows you to link routines written in different programming languages. This facility can lead to unique problems, however, especially when information is passed between routines written in different languages.

To use GSBug to debug parameter-passing problems, use the following procedure:

1. Set breakpoints at the beginning of the calling segment and at the beginning of the called segment.
2. Run the program in trace or real-time mode until it reaches the first breakpoint. Search the calling segment to find the JSL that calls the other segment. If you do not need to interact with the program, the easiest way to do this is to run in trace mode with the Master display on the screen until the second breakpoint is reached. Then both the JSL and the first instruction of the called segment will be on the screen. If you cannot do that, try listing a disassembly (using the `addressL` command) until you see the appropriate JSL.
3. Set a breakpoint just before the JSL that calls the second segment. You can remove the other two breakpoints now if you wish.
4. Run the program until it reaches the JSL breakpoint. Parameters are normally passed either on the stack or in the A, X, and Y registers. The actual information passed may be a pointer to the data rather than the data itself. By examining the contents of the registers, the stack, and memory, determine the location and value of the parameter that is being passed.
5. Execute the JSL. The return address should have been added to the stack.
6. Step through the segment in single-step mode. Is the called routine reading the parameter passed to it? If more than one parameter was passed, are the parameters being read in the correct order? Is an integer being handled as floating point, or is an ASCII string being handled as a number? Is a number being truncated or rounded inappropriately? By a careful study of the action of the called routine, you should be able to determine the source of the problem.
7. If all parameters are being passed correctly, perhaps the problem occurs when the results are passed back to the calling routine. To find the RTL, return to the JSL and start single-step mode. Then type `R`; GSBug will enter trace mode and automatically stop when the next RTL is reached. You might have to do this several times until you reach the right RTL. Study the stack and registers as before to determine whether the results are being correctly passed back to the calling routine.

Part 3: GSBug Subdisplay and Command Reference

This part of the chapter describes each of the subdisplays of the Master display in detail. It presents the commands for customizing the Master display and for setting memory addresses, memory-protection ranges, and breakpoints. It describes the use of the Disassembly subdisplay and all the commands that you can enter on the command line.

Register Subdisplay

The Register subdisplay, along the top of the Master display (see Figure 16.3), shows the contents of both GSBug's and the 65816's registers.

KEY	BRK	Debug	K/PC	B	D	S	A	X	Y	M	Q	L	P	nvmdize	e
00	o d	0100	12/102E	E1	0800	01FD	1F00	3AB9	0007	5F	33	1	30	00110000	0
01FF:23	00/0305:	E1 'a'	E0/100D-03-01	12/1000:	AD 15 18	LDA 1815									
01FE:9D	01/08E0:	9DBF	12/1103-00-00	12/1003:	9D 50 10	STA 1050,X									
01FD:66	01/2003:	121200	00/0000-00-00	12/1006:	9F 20 30 05	STA 053020									
01FC:DA	02/0BEA:	40 '█'	00/0000-00-00	12/100A:	A9 77 66	LDA #6677									
01FB:A3	00/0000:	A0 ' '	00/0000-00-00	12/100D:	82 20 10	BRL 2030 {+1020}									
01FA:39	00/0000:	A0 ' '	00/0000-00-00	12/1010:	80 20	BRA 1032 {+20}									
01F9:5A	00/0000:	A0 ' '	00/0000-00-00	12/1012:	F4 12 34	PEA 3412									
01F8:5A	00/0000:	A0 ' '	00/0000-00-00	12/1015:	62 FC FF	PER 1012 {-0004}									
01F7:5A	00/0000:	A0 ' '	00/0000-00-00	12/1018:	87 45	STA [45]									
01F6:5A	00/0000:	A0 ' '		12/101A:	62 00 F0	PER 001D									
01F5:A9	00/0000:	A0 ' '	E1/0000.000F-T	12/101D:	A9 23	LDA #23									
01F4:39	00/0000:	A0 ' '	01/0900.1FFF-T	12/101F:	A2 45 67	LDX #6745									
01F3:FF	00/0000:	A0 ' '	01/6000.95FF-W	12/1022:	4F 50 4E 02	EOR 024E50									
01F2:3F	00/0000:	A0 ' '	01/D000.D633-W	12/1026:	DC 89 23	JML (2389)									
01F1:23	00/0000:	A0 ' '	00/0000.0000-?	12/1029:	7C BE F2	JSL (F2BE,X)									
01F0:00	00/0000:	A0 ' '	00/0000.0000-?	12/102C:	73 40	ADC (40,S),Y									
01EF:01	00/0000:	A0 ' '	00/0000.0000-?	12/102E:	C1 06	CMP (06),Y									
01EE:A4	00/0000:	A0 ' '	00/0000.0000-?	12/1030:	0A	ASL									
01ED:3E	00/0000:	A0 ' '	00/0000.0000-?	12/1031:	00 23	BRK 23									
COMMAND INPUT LINE															

Figure 16.3: Register Subdisplay Area

Debugger Registers

GSBug's registers, displayed toward the left end of the Register subdisplay, are discussed in the following sections.

Keystroke Modifier

KEY denotes the keystroke modifier. This hexadecimal byte indicates the key or key combination that you can use as a command filter to prevent debugger commands from interfering with your test program when it is running in trace or single-step modes. See "The Command Filter" earlier in this chapter for details.

To select the key or keys to be used as the keystroke modifier, use the command

Key=keynum

Each bit of the binary number represented by the hexadecimal number *keynum* specifies one of the keys to be used as a keystroke modifier; set that bit to 1 to make the key part of the keystroke modifier. Figure 16.4 shows how the bits are assigned from 7 (MSB) to 0 (LSB).

Bit:	7	6	5	4	3	2	1	0
Key:	A	O		K	R	CL	C	S
Hex Value:	80	40	20	10	08	04	02	01

Figure 16.4: Keystroke Modifier Bit Assignments

Table 16.13 shows the keys to which the abbreviations in Figure 16.4 refer.

<u>Abbreviation</u>	<u>Key</u>
S	Shift
C	Control
CL	Caps Lock
R	Repeat (hold the key down until it repeats)
K	Any key on an external keypad (not the keypad on the Apple IIGS keyboard)
O	Option
A	⌘

Table 16.13: Abbreviations in Keystroke Modifier Bit Assignments

For example, to set both the Shift and Caps Lock keys as keystroke modifiers, use the command

```
Key=05
```

The KEY value in the register subdisplay changes to 05 to indicate the key combination that is set as the keystroke modifier (01 for the Shift key plus 04 for the Caps Lock key). Now when you want to send a command to the debugger while in trace or single-step modes press both the Shift and Caps Lock keys while pressing the key that invokes the command. For example, to switch to the slow trace rate, press the key combination

```
Shift-Caps Lock-left-arrow
```

Breakpoint flags

BRK stands for the two breakpoint flags. The first flag reads i (for in) if you have used the debugger to set real breakpoints in the program. If you have transparent breakpoints set, this flag reads o (for out). The second flag reads d (for debugger) if BRK instructions (other than those that you have inserted with the debugger) return you to the debugger. If such BRKs cause an exit to the Monitor, this flag reads m (for monitor). To set user breaks, use the Mbrk command. For details on the i and o flags, see "Breakpoints" earlier in this chapter.

Debugger Direct-Page Indicator

DebugD stands for the debugger's direct page. This value indicates the starting location of the 1KB direct-page/stack block allocated by the debugger in bank \$00 at startup. For instance, if the debugger's direct page begins at 00/1000, DebugD reads 1000.

65816 Registers

Program Bank Register and Program Counter

K/PC stand for the Program Bank register (K) and Program Counter (PC) values. The Program Bank register serves as the upper 8 bits of the 24-bit address of the next instruction to be executed; the program counter holds the lower 16 bits of the address of the next instruction. There is no carry from the high bit of the PC into the low bit of the K register when the PC is

incremented. When you specify an address in a debugger command and do not specify the bank number, the current value of the K register is used.

Data-Bank Register

B denotes the Data-Bank register, which is simply the current value of the data bank. This value provides the upper 8 bits of the address in addressing modes that generate only the lower 16 bits.

Direct-Page Register

D indicates the Direct-Page register, which holds the current direct page. This value determines the location of the direct page in bank \$00. For example, if the direct page begins at 00/1234, D reads 1234.

Stack Pointer

S shows the current value of the Stack pointer. This register indicates the next available location on the stack. If the emulation-mode flag $e = 1$, S must be between \$0100 and \$01FF.

Accumulator

A displays the current value of the Accumulator, which stores first one operand and then the result for most arithmetic and logical operations. This register is 2 bytes wide if the emulation-mode flag $e = 0$ and the memory/accumulator-mode flag $m = 0$; otherwise, it is considered to be 1 byte wide (though the high byte can still be accessed through an XBA instruction). The Accumulator display is always 16 bits, even in emulation mode or when 8-bit indexing is selected.

X Register

X indicates the X register display, which shows the current value of the X register. This register provides index values for address calculations and holds operands for some arithmetic and logical operations. This register is 2 bytes wide if $e = 0$ and the index-register-mode flag $x = 0$; otherwise, it is considered to be 1 byte wide (the high byte is forced to 0). The X register display is always 16 bits, even in emulation mode or when 8-bit indexing is selected.

Y Register

Y denotes the Y register display which shows the current value of the Y register. This register provides index values for address calculations, and it holds operands for some arithmetic and logical operations. This register is 2 bytes wide if $e = 0$ and $x = 0$; otherwise, it is considered to be 1 byte wide (the high byte is forced to 0). The Y register display is always 16 bits, even in emulation mode or when 8-bit indexing is selected.

Machine-State Register

M stands for the Machine-State register. This pseudoregister, located at \$C068 (in any of banks \$00, \$01, \$E0, or \$E1), can be used to set a variety of Mega II-chip soft switches. The M

register is described in detail in the Apple IIGS Hardware Reference. Table 16.14 gives the bits that comprise this hexadecimal number to indicate the status of the machine states.

<u>Bit</u>	<u>Name</u>	<u>Meaning</u>
7	ALTZP	If this bit is 0, bank-switched memory, stack, and zero page are in main memory; if it is 1, they are in auxiliary memory.
6	PAGE2	If this bit is 0, text page 1 is selected; if it is 1, text page 2 is selected.
5	RAMRD	If this bit is 0, main-memory RAM is read-enabled; if it is 1, auxiliary-memory RAM is read-enabled.
4	RAMWRT	If this bit is 0, main-memory RAM is write-enabled; if it is 1, auxiliary-memory RAM is write-enabled.
3	RDRAM	If this bit is 0, language-card RAM is read-enabled; if it is 1, language-card ROM is read-enabled.
2	LCBNK2	If this bit is 0, bank 2 language-card RAM (at \$D000 through \$DFFF) is selected; if it is 1, bank 1 language-card RAM is selected. Switching banks with this bit does not write-enable language-card RAM. Use the L flag both to write-enable RAM and to switch language-card banks.
1	ROMBANK	This bit is reserved; it must equal 0.
0	INTCXROM	If this bit is 0, external ROM (that is, ROM on the circuit board at \$Cx00) is active; if it is 1, internal ROM at \$Cx00 is active.

Table 16.14: Status bits for Machine-State register

Setting ROMBANK to 1 will almost certainly cause the system to crash.

Quagmire Register

Q stands for the Quagmire register. This pseudoregister is composed of the lower seven bits of the Shadow register at \$C035 and the high bit of the Configuration register at \$C036. The bits of this hexadecimal number have the meanings given in Table 16.15.

<u>Bit</u>	<u>Meaning</u>
7	If this bit is 1, high-speed operation is on.
6	If this bit is 1, IOLC (I/O and language card) shadowing is off.
5	This bit is reserved; it must equal 1.
4	If this bit is 1, auxiliary-memory Hi-Res graphics shadowing is off.
3	If this bit is 1, Super Hi-Res graphics shadowing is off.
2	If this bit is 1, Hi-Res graphics page 2 shadowing is off.
1	If this bit is 1, Hi-Res graphics page 1 shadowing is off.
0	If this bit is 1, text page 1 shadowing is off.

Table 16.15: Status bits of the Quagmire register

Shadowing is described in the Technical Introduction to the Apple IIGS and the Apple IIGS Hardware Reference.

Language-Card Bank Flag

L stands for the language-card bank flag, which indicates whether bank 1 or bank 2 of the language card is selected. This flag emulates the Monitor value=L command. Set L to 0 to write-enable language-card RAM and select bank 1. Set L to 1 to write-enable language-card RAM and select bank 2. Changing L automatically changes bit 2 of the M register.

Processor-Status Register

P stands for the Processor-Status register, which holds status flags and mode-select bits. The bits and flags appear in two forms. Directly under the P, the information appears as a hexadecimal number. At the right end of the Register subdisplay, each individual bit is labeled and shown in binary. Figure 16.5 shows the meanings of these bits in emulation mode.

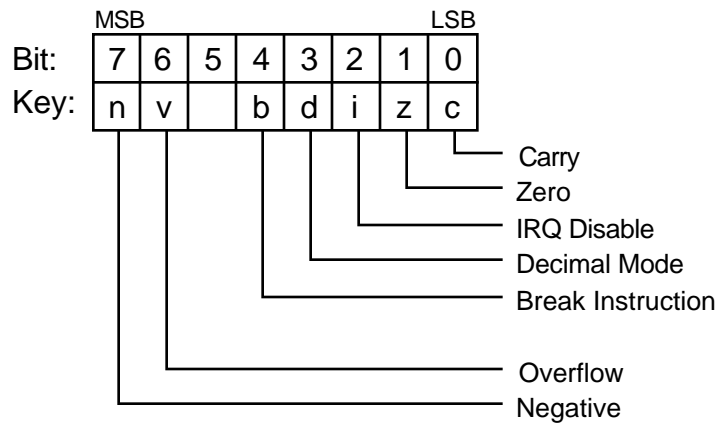


Figure 16.5: Processor-Status Register Bits in Emulation Mode

Figure 16.6 shows the meanings of the bits in native mode.

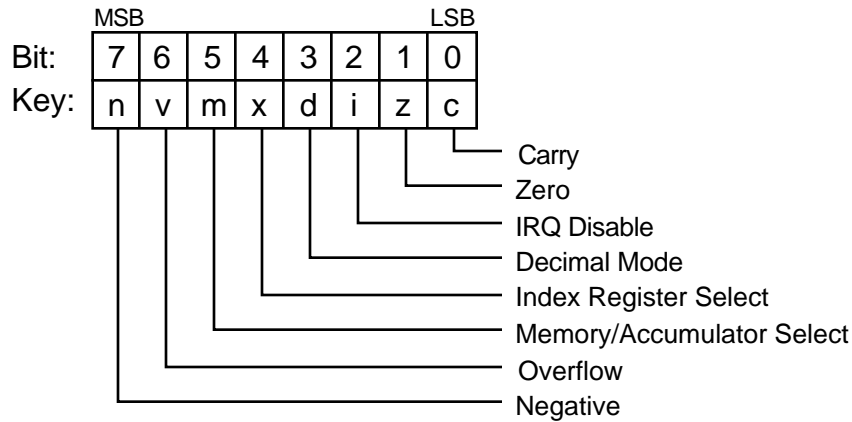


Figure 16.6: Processor-Status Register Bits in Native Mode

Emulation Mode Flag

E stands for the emulation mode flag. If $e = 1$, 6502 emulation mode is selected. If $e = 0$, then native mode is selected. See Figures 16-5 and 16-6 for the meanings of the Processor-Status register bits in emulation and native modes.

Disassembly Mode Flag

D stands for the disassembly mode flag. It indicates which disassembly mode is used during the List command. If d equals 0, the debugger recognizes REP and SEP instructions to change the accumulator and index sizes and displays those instructions that follow the REP and SEP accordingly. If d equals 1, the debugger ignores REP and SEP instructions and displays all instructions in the format determined by the m and x bits of the Processor-Status register.

Altering the Contents of Registers

To alter the contents of some of the registers and flags displayed in the Master display, type one of the commands in Table 16.16 on the command line and press Return.

To set the A, X, Q, Y, and L registers to the value specified by value, observe the following rules:

- The values for all registers are given as hexadecimal numbers, except for the processor-status bits, which can be either 1 or 0.
- Register names are case sensitive. For example, to set the X index register to \$12E0, use the command

```
X=12E0
```

The commands in the following table are case sensitive.

<u>Command</u>	<u>Action</u>
<i>K=bank</i>	Sets K (Program bank register) to <i>bank</i> .
<i>PC=address</i>	Sets PC (Program counter) to <i>address</i> (sets K to bank if <i>address</i> is long).
<i>K/PC=address</i>	Sets PC (Program counter) to <i>address</i> (sets K to bank if <i>address</i> is long).
<i>B=bank</i>	Sets B (Data bank register) to <i>bank</i> .
<i>D=address</i>	Sets D (Direct-Page register) to <i>address</i> .
<i>DPAGE</i>	Sets D (Direct-Page register) to the direct page the debugger allocated for the user at startup.
<i>S=address</i>	Sets the Stack pointer to <i>address</i> . The direct page and stack are always in bank zero, so you do not need to specify a bank when you alter the contents of the Direct-Page register or the Stack pointer.
<i>STACK</i>	Sets S to the stack pointer that the debugger allocated for the user at startup.
<i>A=value</i>	Sets A (Accumulator) to <i>value</i> .
<i>X=value</i>	Sets X (X register) to <i>value</i> .
<i>Y=value</i>	Sets Y (Y register) to <i>value</i> .
<i>M=value</i>	Sets M (Machine-state register) to <i>value</i> .
<i>Q=value</i>	Sets Q (Quagmire register) to <i>value</i> .
<i>L=value</i>	Sets L (Language-card bank flag) to <i>value</i> , where <i>value</i> = 0 or 1. If a greater value is entered, only its least significant bit is used.
<i>P=value</i>	Sets P (Processor-Status register) to <i>value</i> .
<i>x</i> [lower case]	Toggles the x bit of the Processor-Status register (only in 65816 mode).
<i>m</i> [lower case]	Toggles the m bit of the Processor-Status register (only in 65816 mode). If the x or m commands are entered while 6502 emulation is on, the debugger will beep to indicate an error.
<i>e</i>	Toggles the e (Emulation mode) flag: if this flag is set to 1, changes it to 0; if it's set to 0, changes it to 1.
<i>m</i>	Toggles the m flag: if this flag is set to 1, changes it to 0; if it's set to 0, changes it to 1. This command works only if e = 0.

Table 16.16: Commands to Alter Contents of Registers in the Master Display

The lengths of the X, Y, and A registers depend on the settings of the e, x, and m processor bits, as discussed in the register descriptions in the preceding section, "65816 Registers."

Stack Subdisplay

KEY	BRK	DeBugD	K/PC	B	D	S	A	X	Y	M	Q	L	P	nvmd12c	e
00	o	d	0100	12/102E	E1	0800	01FD	1F00	3AB9	0007	5F	33	1	30	00110000 0
01FF:23	00/0305:	E1 'a'	E0/100D-03-01	12/1000:	AD 15 18	LDA 1815									
01FE:9D	01/08E0:	9DBF	12/1103-00-00	12/1003:	9D 50 10	STA 1050,X									
01FD:66	01/2003:	121200	00/0000-00-00	12/1006:	9F 20 30 05	STA 053020									
01FC:DA	02/0BEA:	40 '█'	00/0000-00-00	12/100A:	A9 77 66	LDA #6677									
01FB:A3	00/0000:	A0 ' '	00/0000-00-00	12/100D:	82 20 10	BRL 2030 {+1020}									
01FA:39	00/0000:	A0 ' '	00/0000-00-00	12/1010:	80 20	BRA 1032 {+20}									
01F9:5A	00/0000:	A0 ' '	00/0000-00-00	12/1012:	F4 12 34	PEA 3412									
01F8:5A	00/0000:	A0 ' '	00/0000-00-00	12/1015:	62 FC FF	PER 1012 {-0004}									
01F7:5A	00/0000:	A0 ' '	00/0000-00-00	12/1018:	87 45	STA [45]									
01F6:5A	00/0000:	A0 ' '		12/101A:	62 00 F0	PER 001D									
01F5:A9	00/0000:	A0 ' '	E1/0000.000F-T	12/101D:	A9 23	LDA #23									
01F4:39	00/0000:	A0 ' '	01/0900.1FFF-T	12/101F:	A2 45 67	LDX #6745									
01F3:FF	00/0000:	A0 ' '	01/6000.95FF-W	12/1022:	4F 50 4E 02	EOR 024E50									
01F2:3F	00/0000:	A0 ' '	01/D000.D633-W	12/1026:	DC 89 23	JML (2389,X)									
01F1:23	00/0000:	A0 ' '	00/0000.0000-?	12/1029:	7C BE F2	JSL (F2BE,X)									
01F0:00	00/0000:	A0 ' '	00/0000.0000-?	12/102C:	73 40	ADC (40,S),Y									
01EF:01	00/0000:	A0 ' '	00/0000.0000-?	12/102E:	C1 06	CMP (06),Y									
01EE:A4	00/0000:	A0 ' '	00/0000.0000-?	12/1030:	0A	ASL									
01ED:3E	00/0000:	A0 ' '	00/0000.0000-?	12/1031:	00 23	BRK 23									

Figure 16.7: Stack Subdisplay Area

The Stack subdisplay, along the far left side of the Master display, shows the contents of a portion (19 bytes) of the 65816's stack. This subdisplay shows the addresses and contents of the memory locations just before and just after the location pointed to by the stack pointer. The current location of the stack pointer is shown in the Register subdisplay (see the earlier section, "Register Subdisplay") and is highlighted in the Stack subdisplay. The stack address can appear as an absolute address in bank zero or as an offset from the top of the stack.

Here's an example of a typical line in the stack subdisplay showing absolute addresses.

```
00AF1:57
```

In this instance, the address is \$0AF1 (the stack is always in bank zero), and the contents of the stack location is \$5F.

Here's an example of a typical line in the stack subdisplay showing stack offsets.

```
+05 :D2
```

In this instance, \$D2 is stored five bytes above the top of the stack.

The stack address is initially formatted as an absolute address. You can change this format to stack offsets with the Set command. See "Configuring the Master Display" for details on the Set command.

The Stack pointer points to the current stack location, which is always highlighted. As the stack grows downward in memory, the Stack subdisplay scrolls up so the current stack location remains in the same place on the screen. In a similar fashion, the display scrolls down as the stack shrinks.

The debugger initially places the current stack location at the bottom of the subdisplay. You can change the position of the current stack location within this subdisplay by using the Set command. See the section "Configuring the Master Display" later in this chapter for a discussion of the Set command.

See the section "RAM Subdisplay" later in this chapter for a discussion of commands you can use to change values in memory.

Disassembly Subdisplay

KEY	BRK	DeBugD	K/PC	B	D	S	A	X	Y	M	Q	L	P	nvmd12c	e
00	o d	0100	12/102E	E1	0800	01FD	1F00	3AB9	0007	5F	33	1	30	00110000	0
01FF:23	00/0305:	E1 'a'	E0/100D-03-01	12/1000:	AD 15 18	LDA 1815									
01FE:9D	01/08E0:	9DBF	12/1103-00-00	12/1003:	9D 50 10	STA 1050,X									
01FD:66	01/2003:	121200	00/0000-00-00	12/1006:	9F 20 30 05	STA 053020									
01FC:DA	02/08EA:	40 '█'	00/0000-00-00	12/100A:	A9 77 66	LDA #6677									
01FB:A3	00/0000:	A0 ' '	00/0000-00-00	12/100D:	82 20 10	BRL 2030 {+1020}									
01FA:39	00/0000:	A0 ' '	00/0000-00-00	12/1010:	80 20	BRA 1032 {+20}									
01F9:5A	00/0000:	A0 ' '	00/0000-00-00	12/1012:	F4 12 34	PEA 3412									
01F8:5A	00/0000:	A0 ' '	00/0000-00-00	12/1015:	62 FC FF	PER 1012 {-0004}									
01F7:5A	00/0000:	A0 ' '	00/0000-00-00	12/1018:	87 45	STA [45]									
01F6:5A	00/0000:	A0 ' '		12/101A:	62 00 F0	PER 001D									
01F5:A9	00/0000:	A0 ' '	E1/0000.000F-T	12/101D:	A9 23	LDA #23									
01F4:39	00/0000:	A0 ' '	01/0900.1FFF-T	12/101F:	A2 45 67	LDX #6745									
01F3:FF	00/0000:	A0 ' '	01/6000.95FF-W	12/1022:	4F 50 4E 02	EOR 024E50									
01F2:3F	00/0000:	A0 ' '	01/D000.D633-W	12/1026:	DC 89 23	JML (2389)									
01F1:23	00/0000:	A0 ' '	00/0000.0000-2	12/1029:	7C BE F2	JSL (F2BE,X)									
01F0:00	00/0000:	A0 ' '	00/0000.0000-2	12/102C:	73 40	ADC (40,S),Y									
01EF:01	00/0000:	A0 ' '	00/0000.0000-2	12/102E:	C1 06	CMP (06),Y									
01EE:A4	00/0000:	A0 ' '	00/0000.0000-2	12/1030:	0A	ASL									
01ED:3E	00/0000:	A0 ' '	00/0000.0000-2	12/1031:	00 23	BRK 23									

Figure 16.8: Disassembly Subdisplay Area

The Disassembly subdisplay along the right side of the Master display shows a disassembly of up to 19 lines of your program's object code using standard 65816 assembly mnemonics and address-mode syntax. The debugger shows the 65816 disassembly while executing code in single-step or trace mode and while listing code.

As shown in Figure 16.9, each line of this subdisplay is composed of three parts: the address (in bank/location format), the bytes of the instruction at that address that make up the instruction, and the disassembled version of those bytes.

For example, look at the first line of the Disassembly subdisplay in Figure 16.9. The first part of the line, the address, is composed of the high-order byte (\$12) that specifies the memory bank, followed by the 2-byte (\$1000) location within that bank of the first byte in the instruction.

```

12/1000: AD 15 18    LDA 1815
12/1003: 9D 50 10    STA 1050,X
12/1006: 9F 20 30 05 STA 053020
12/100A: A9 77 66    LDA #6677
12/100D: 82 20 10    BRL 2030 {+1020}
12/1010: 80 20      BRA 1032 {+20}
12/1012: F4 12 34    PEA 3412
12/1015: 62 FC FF    PER 1012 {-0004}
12/1018: 87 45      STA [45]
12/101A: 62 00 F0    PER 001D
12/101D: A9 23      LDA #23
12/101F: A2 45 67    LDX #6745
12/1022: 4F 50 4E 02 EOR 024E50
12/1026: DC 89 23    JML (2389)
12/1029: 7C BE F2    JSL (F2BE,X)
12/102C: 73 40      ADC (40,S),Y
12/102E: C1 06      CMP (06),Y
12/1030: 0A        ASL
12/1031: 00 23      BRK 23

```

Figure 16.9: Columns of the Disassembly Subdisplay

The next 1 to 4 bytes (AD 15 18 in this example) show the contents of memory starting at that location, corresponding to the instruction that GSDebug interpreted to start at that location. The

last part of the line (LDA 1815) shows the 65816 opcode and operand. The addressing mode syntax (operand address formats) summarized in Table 16.17 is the same as that used by the Apple IIGS Monitor Mini-Assembler and described in the Apple IIGS Firmware Reference. All numbers are hexadecimal.

<u>Addressing Mode</u>	<u>Example Operand</u>
Absolute	1234
Absolute indexed with X	1234,X
Absolute indexed with Y	1234,Y
Absolute indexed indirect	(1234,X)
Absolute indexed long	081234,X
Absolute indirect long	(1234)
Absolute long	081234
Accumulator*	
Block move	5678
Direct page	12
Direct page indexed with X	12,X
Direct page indexed with Y	12,Y
Direct page indexed indirect with X	(12,X)
Direct page indirect	(12)
Direct page indirect indexed with Y	(12),Y
Direct page indirect indexed long	[12]
Direct page indirect long	[12]
Immediate	#12 or #1234
Implied*	
Program counter relative	1000 {+12}
Program counter relative long	1000 {-1234}
Stack*	
Stack relative	10,S
Stack relative indirect indexed with Y	(10,S),Y

*These addressing modes require no operand.

Table 16.17: Disassembly Operand Formats

You can change the position of the current instruction within the subdisplay by using the Set command. See the section "Configuring the Master display" later in this chapter for a discussion of the Set command.

The GSbug disassembler interprets all bytes in memory as 65816 instructions; it cannot distinguish between code and data. Strings of data therefore appear as nonsense instructions in the Disassembly subdisplay.

When you start the debugger, the Disassembly subdisplay is blank. The debugger enters values into this subdisplay in response to any of the command-line commands shown in Table 16.18.

Whenever you enter an address or a value, the debugger assumes it is in hexadecimal. You can precede the address or a value with a dollar sign or not as you wish. For long addresses you can include the slash after the bank (for example, 12/3456) or not, as you wish. If you do not include a bank number (or enter a short address of two bytes or less), the current value of the K register is used for the bank.

<u>Command</u>	<u>Action</u>
<i>address</i> L	Disassembles the contents of memory starting at <i>address</i> , and the next 19 lines (one screen full) of that disassembly appear in the subdisplay.
L	Displays the next 19 lines of the disassembly.
<i>address</i> T	Enters trace mode at <i>address</i> . If you omit <i>address</i> , uses the current K/PC address. As GSBug traces the code, it disassembles it and lists the results in the Disassembly subdisplay. The currently executing instruction is highlighted. Trace mode is described in the section "Single-Step and Trace Modes" earlier in this chapter.
<i>address</i> S	Enters single-step mode at <i>address</i> . If you omit <i>address</i> , uses the current K/PC address. As GSBug steps through the code, it disassembles it and lists the results in the Disassembly subdisplay. The currently executing instruction is highlighted. Single-step mode is described in the section "Running Your Program" earlier in this chapter.

Table 16.18: Commands for Entering Values into Disassembly Subdisplay

Entering Instructions into the Mini-Assembler Display

The mini-assembler lets you enter machine-language programs directly from the keyboard using standard mnemonics. To turn on the built-in Mini-assembler display, use the command

Asm

and press Return. The Master display is turned on and any instructions you enter appear at the bottom of the Disassembly subdisplay as you type them in.

Table 16.19 summarizes the syntax for entering instructions into memory to be assembled.

<u>Command</u>	<u>Action</u>
Asm	Clears the Disassembly subdisplay to prepare to enter a sequence of instructions using the address:instruction command.
<i>address:instruction</i>	Causes the debugger to assemble the instruction <i>instruction</i> and place the 65816 opcode and operand in memory at <i>address</i> . Simultaneously, it places the instruction on the last line of the Disassembly subdisplay. Use the Asm command before using this command. See the section "Altering the Contents of Memory" later in this chapter for more information on this command and a discussion of other commands to change values in memory.
Space bar <i>instruction</i>	Once you have used the <i>address:instruction</i> command, this command causes the next available address to be printed on the command line. Type in the next instruction to be assembled and press Return.

Table 16.19: Mini-Assembler Commands

Using the Mini-Assembler is the same as entering mon, typing ! and then entering instructions.

Determining Instruction Length

GSBug determines the length of an instruction entered into memory entirely by the length of the operand that is entered, not by the current value of any flags. Table 16.20 provides some examples of how the debugger calculates the length of an instruction.

<u>Instruction</u>	<u>Bytes Entered Into Memory</u>
JMP 1234	\$4C \$34 \$12
JMP 123456	\$5C \$56 \$34 \$12
LDA F1	\$A4 \$F1
LDA 00F1	\$A4 \$F1 \$00

Table 16.20: Instruction Length

In addition, the length of the operand must have the exact number of bytes required. For instance, the instruction

```
JSL 3000
```

is not legal. The correct instruction would be

```
JSL 003000
```

Displaying Toolbox Instructions and GS/OS Calls

GSBug also recognizes and displays two special instructions: toolbox calls and GS/OS calls. When the debugger encounters a toolbox call (that is, JSL \$E10000) in a disassembly, it finds out if the preceding instruction was an LDX. If so, the actual tool call name, preceded by an underscore, appears instead of the JSL instruction. For example, the instructions

```
LDX #0101
JSL E10000
```

appear in the subdisplay as

```
LDX #0101
_TLBootInit
```

When the debugger encounters a tool call that it does not recognize, it replaces the JSL E10000 with _Unknown. If the previous instruction was not an LDX, the debugger displays the JSL as usual.

When the debugger encounters a GS/OS call (that is, JSL \$E100A8), it displays JSL GS/OS instead. It then interprets the next two bytes as the GS/OS call number and shows the call name. If it does not recognize the call number, Unknown appears. The debugger then interprets the following four bytes as the address of the parameter list and shows them as a long address. Here is how a typical GS/OS call might be displayed:

Shell Reference Manual

```
01/6000: 22    A8    00    E1    JSL    GS/OS
01/6004: 01    00
01/6006: 56    34    12    00    CREATE
                                $12/3456
```

Displaying Tool Call Information

You can also request that the debugger display tool call names and tool call numbers along with the address of the actual tool routine with the Tool commands.

To display the name, number, and address of tool number *toolnum*, use the command

```
Tool #toolnum
```

To display the name, number, and address of tool *toolname*, use the command

```
Tool _toolname
```

For example, type

```
Tool #0101
```

to display the information

```
Tool #0101 TLBootInit @11/860A
```

which tells you that tool number 0101 is called TLBootInit and the actual routine is located at \$11/860A. This information appears on the command line and disappears with the next keystroke.

If you type in an unknown tool number with a Tool command, Unknown appears instead of the tool name. If you type in an unknown tool name, the debugger does not accept the command but simply beeps.

Displaying Instructions During Trace and Single-Step Modes

During trace and single-step modes, the debugger highlights the instruction that is about to be executed. While the code is being executed, it scrolls up the screen so that the current instruction remains in the same location. The current instruction initially appears one line up from the bottom of the subdisplay. You can change this location with the Set command. See "Configuring the Master Display" for more details.

The Disassembly subdisplay shows the instructions following the current instruction in memory. If the current instruction jumps to or calls another routine, however, the address called or jumped to appears as the current instruction after the call or jump is executed. For example, assume the Disassembly subdisplay is

```
12/102E:C1 06    CMP (06),Y
12/1030:0A      ASL
12/1031:DC 89 23  JML 012389
12/1034:7C BE F2  JSL (F2BE,X)
12/1037:AD 15 18  LDA (1815)
```


When the JML instruction is executed, the current instruction becomes the one jumped to, and the display changes accordingly. It might look like this:

```
12/1030:0A          ASL
12/1031:DC 89 23    JML 012389
01/2389:9D 50 10    STA 1050,X
01/238C:9F 20 30 05 STA 053020
01/2390:A9 77 66    LDA #6677
```

RAM Subdisplay

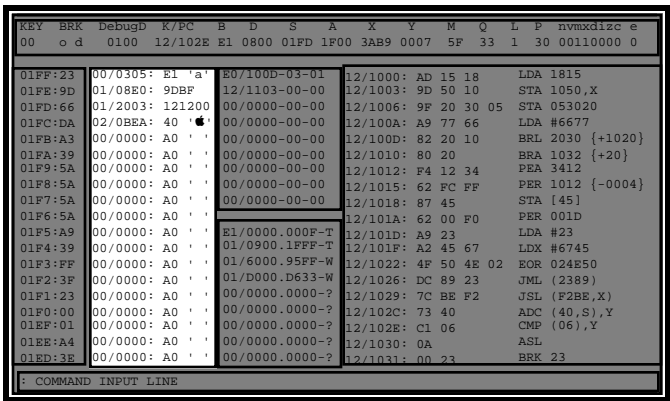


Figure 16.10: RAM Subdisplay Area

The RAM subdisplay, located to the right of the Stack subdisplay, shows the contents of any 19 memory locations you select (one per line). Each line displays an address followed by the contents of the specified memory. Here is a typical line in the RAM subdisplay.

```
E0/6298: 63F9
```

Each line is displayed as either an absolute or a direct-page address. The debugger represents absolute addresses in the form of bank/location. A direct-page address includes the letters DP followed by a 1-byte direct-page offset.

The absolute address of a direct-page address changes if the direct-page register changes.

Each address can be referenced either indirectly or directly. The debugger supports indirect referencing by displaying the contents of memory pointed to by the address on a given line. A single colon after an address indicates direct referencing.

Indirect referencing (also called dereferencing) occurs when the screen shows the contents of the memory pointed to by the address stored at the address shown on the specified line.

The pointer stored at the specified address can be either a 2-byte or a 3-byte address. Two colons following the address represent 2-byte dereferencing while three colons following the address represent 3-byte dereferencing. In 2-byte dereferencing the current value of the Data-Bank register is used as the bank. Both 2-byte and 3-byte pointers must be stored with their least significant bytes first.

In Figure 16.11 each location can show either a single hexadecimal byte value and the equivalent ASCII character, a 2-byte value, or a 3-byte value. The 2- and 3-byte values are displayed as addresses; that is, the low byte (the one corresponding to the address in the left column) is displayed at the right. For example, if you place the value 1A in location 01/0100, 1B in 01/0101, and 1C in 01/0102, and display a 3-byte value at 01/0100, the line of the RAM subdisplay looks like

01/0100: 1C1B1A

Type : to toggle the number of colons following an address.

[illegible]

Figure 16.11: Columns of the RAM Subdisplay

To modify the contents of the RAM subdisplay, type **MEM** on the command line and press Return. The cursor appears on the highlighted first line in the subdisplay. You can now use any of the commands listed in Table 16.21. (To enter specific values into memory locations, use the commands described in the section "Altering the Contents of Memory" later in this chapter.)

<u>Press</u>	<u>Action</u>
Return key	Moves down one address (if on bottom address, wraps to top).
down-arrow	Moves down one address (if on bottom address, wraps to top).
up-arrow	Moves up one address (if on top address, wraps to bottom).
hexadecimal_digit	Enters the address.
H	Displays the contents of memory as hexadecimal and ASCII.
P	Displays the contents of memory as a 2-byte address.
L	Displays the contents of memory as a 3-byte address.
Z	Toggles between direct-page and absolute address.
:	Toggles between direct, 2-byte indirect, and 3-byte indirect addressing.
?	Displays Memory subdisplay Help screen. Press any key except Esc to return to the RAM subdisplay.
Esc	Returns to the command line.

Table 16.21: Commands for Editing the RAM Subdisplay

Breakpoint Subdisplay

KEY	BRK	Debug	K/PC	B	D	S	A	X	Y	M	Q	L	P	nvmdize	e
00	o d	0100	12/102E	E1	0800	01FD	1F00	3AB9	0007	5F	33	1	30	00110000	0
01FF:23		00/0305:	E1 'a'				E0/100D-03-01								
01FE:9D		01/08E0:	9DBF				12/1103-00-00								
01FD:66		01/2003:	121200				00/0000-00-00								
01FC:DA		02/08EA:	40 'A'				00/0000-00-00								
01FB:A3		00/0000:	A0 'A'				00/0000-00-00								
01FA:39		00/0000:	A0 'A'				00/0000-00-00								
01F9:5A		00/0000:	A0 'A'				00/0000-00-00								
01F8:5A		00/0000:	A0 'A'				00/0000-00-00								
01F7:5A		00/0000:	A0 'A'				00/0000-00-00								
01F6:5A		00/0000:	A0 'A'				00/0000-00-00								
01F5:A9		00/0000:	A0 'A'				E1/0000.000F-T								
01F4:39		00/0000:	A0 'A'				01/0900.1FFF-T								
01F3:FF		00/0000:	A0 'A'				01/6000.95FF-W								
01F2:3F		00/0000:	A0 'A'				01/D000.D633-W								
01F1:23		00/0000:	A0 'A'				00/0000.0000-?								
01F0:00		00/0000:	A0 'A'				00/0000.0000-?								
01EF:01		00/0000:	A0 'A'				00/0000.0000-?								
01EE:A4		00/0000:	A0 'A'				00/0000.0000-?								
01ED:3E		00/0000:	A0 'A'				00/0000.0000-?								

COMMAND INPUT LINE

Figure 16.12: Breakpoint Subdisplay Area

GSDbug allows you to set from 0 to 17 breakpoints in your program. When you set a breakpoint, you indicate the location at which you want the program to suspend execution and the number of times you want the breakpoint to be encountered before execution is interrupted. Each line of the Breakpoint subdisplay shows a breakpoint address (bank/location in bank), the number of times through the breakpoint before it triggers, and the number of times the program has actually passed through this breakpoint so far.

The initial Master display configuration provides nine lines for breakpoints. You can delete breakpoint lines, thus increasing the number of memory-protection lines, or you can delete memory-protection lines to increase the number of breakpoint lines. (The Memory-Protection subdisplay is directly below the Breakpoint subdisplay.)

Trigger values are 1-byte hexadecimal values. A trigger value of \$00 instructs the debugger to ignore the breakpoint. You can use this value to turn off an individual breakpoint.

Shell Reference Manual

For example, the first line in the Breakpoint subdisplay above is

```
E0/100D-03-01
```

This line indicates that GSbug is set to suspend execution of your program the third time it encounters the instruction located at address 100D in bank E0, and that it has already executed this instruction one time.

Interpreted breaks occur during tracing or single-stepping through code. Real breaks occur during real-time execution of code.

Real breakpoints actually replace the instruction at the breakpoint with a BRK instruction. When the debugger encounters a real breakpoint, it compares the count against the trigger value. If they are equal, the debugger halts execution. If not, the debugger executes the real instruction it saved when it was replaced with the BRK instruction.

If you edit the Breakpoint subdisplay when real breakpoints are in, the debugger automatically takes them out when you enter the subdisplay and puts them back in when you exit.

You can type in the breakpoint commands listed in Table 16.22 from the Master display command line. Press Return after typing each of these commands.

<u>Command</u>	<u>Action</u>
Clr	Zeros all breakpoints to 00/0000-00-00. This command does not remove breakpoint lines from the screen. Use the Delete key as described in the preceding description of the Bp command to remove breakpoint lines.
In	Inserts real breakpoints (that is, replaces the instructions at the breakpoints with break instructions). The Brk register changes from o to i, and BRK instructions are inserted in memory at the addresses specified by the Breakpoint subdisplay. You must insert real breakpoints in the code in order to make the debugger suspend execution in real-time mode. Real and interpreted breakpoints are discussed in the section "Breakpoints" earlier in this chapter. You cannot edit the Breakpoint subdisplay when real breakpoints are in. Use the Out command before attempting to change breakpoints or trigger values.
Out	Removes real breakpoints. The Brk register changes from i to o, and the Brk instructions that were inserted in memory by the In command are replaced with interpreted breakpoints. Real and interpreted breakpoints are discussed in the section "Breakpoints" earlier in this chapter. Be sure to remove real breakpoints with the Out command before exiting the debugger. If they are not removed, the break instructions remain in memory and the debugger cannot restore the original instructions.
Dbrk	Returns to the debugger when a BRK instruction that has not been set as a breakpoint is encountered while in real-time mode (or anytime while the init version is installed). A d appears next to the i or o in the Brk register display. DBRK is the initial setting.
Mbrk	Exits to the Monitor when a BRK instruction that has not been set as a breakpoint is encountered while your program is running in real-time mode or anytime while the init version is installed. An m appears next to the i or o in the Brk register display.

Table 16.22: Breakpoint Commands Entered from the Master Display

To increase the number of breakpoints, you must delete the memory-protection ranges. For details, see "Memory-Protection Subdisplay" later in this chapter.

Altering the Contents of the Breakpoint Subdisplay

To alter the contents of the Breakpoint subdisplay, type the following command and press Return:

Bp

You can now use any of the single-keystroke commands shown in Table 16.23. When editing breakpoints, the cursor flashes to indicate the current position within the display.

<u>Command</u>	<u>Action</u>
Return	Moves to the next address down.
down-arrow	Moves to the next address down (if the last line on display, wraps to top).
up-arrow	Moves to the next address up (if the first line on display, wraps to top).
left-arrow	Moves left to the address. Type in the starting address of the instruction at which you want the debugger to suspend execution. You can include a slash (/) after the bank value or omit it when entering the address; either form works. If you do not include the bank number, the current value of the K register is used for the bank.
right-arrow	Moves right to the trigger value. Type in a 1-byte hexadecimal number to indicate the number of times the debugger should execute this instruction before suspending execution. If the value is 0, the debugger ignores the breakpoint. If the value is 1, the debugger stops each time it encounters the breakpoint. If the value is any number n from 2 to 255, the debugger stops every nth time it encounters the breakpoint.
<i>hexadecimal_digit</i>	Types in address or trigger value as a hexadecimal value.
Delete	Deletes the currently highlighted breakpoint and increases the number of memory-protection lines by one.
?	Displays a help screen. Press any key except Esc to return to the Breakpoint subdisplay.
Esc	Returns to the command line.

Table 16.23: Commands for Altering the Contents of the Breakpoint Subdisplay

Memory Protection Subdisplay

KEY	BRK	DeBugD	K/PC	B	D	S	A	X	Y	M	Q	L	P	nvmd12c	e
00	o d	0100	12/102E	E1	0800	01FD	1F00	3AB9	0007	5F	33	1	30	00110000	0
01FF:23	00/0305:	E1 'a'	E0/100D-03-01	12/1000:	AD 15 18	LDA 1815									
01FE:9D	01/08E0:	9DBF	12/1103-00-00	12/1003:	9D 50 10	STA 1050,X									
01FD:66	01/2003:	121200	00/0000-00-00	12/1006:	9F 20 30 05	STA 053020									
01FC:DA	02/08EA:	40 '█'	00/0000-00-00	12/100A:	A9 77 66	LDA #6677									
01FB:A3	00/0000:	A0 ' '	00/0000-00-00	12/100D:	82 20 10	BRL 2030 {+1020}									
01FA:39	00/0000:	A0 ' '	00/0000-00-00	12/1010:	80 20	BRA 1032 {+20}									
01F9:5A	00/0000:	A0 ' '	00/0000-00-00	12/1012:	F4 12 34	PEA 3412									
01F8:5A	00/0000:	A0 ' '	00/0000-00-00	12/1015:	62 FC FF	PER 1012 {-0004}									
01F7:5A	00/0000:	A0 ' '	00/0000-00-00	12/1018:	87 45	STA [45]									
01F6:5A	00/0000:	A0 ' '		12/101A:	62 00 F0	PER 001D									
01F5:A9	00/0000:	A0 ' '	E1/0000.000F-T	12/101D:	A9 23	LDA #23									
01F4:39	00/0000:	A0 ' '	01/0900.1FFF-T	12/101F:	A2 45 67	LDX #6745									
01F3:FF	00/0000:	A0 ' '	01/6000.95FF-W	12/1022:	4F 50 4E 02	EOR 024E50									
01F2:3F	00/0000:	A0 ' '	01/D000.D633-W	12/1026:	DC 89 23	JML (2389)									
01F1:23	00/0000:	A0 ' '	00/0000.0000-?	12/1029:	7C BE F2	JSL (F2BE,X)									
01F0:00	00/0000:	A0 ' '	00/0000.0000-?	12/102C:	73 40	ADX (40,S),Y									
01EF:01	00/0000:	A0 ' '	00/0000.0000-?	12/102E:	C1 06	CMF (06),Y									
01EE:A4	00/0000:	A0 ' '	00/0000.0000-?	12/1030:	0A	ASL									
01ED:3E	00/0000:	A0 ' '	00/0000.0000-?	12/1031:	00 23	BRK 23									

Figure 16.13: Memory-Protection Subdisplay Area

GSBug allows you to specify address ranges that are protected during execution in trace or single-step modes. Each address range you have protected is shown in the Memory-Protection subdisplay, followed by a code that indicates the type of protection set, as shown in Table 16.24.

Code	Meaning
T	Code-trace range. All code outside this range is executed in single-step or trace mode. When your code executes a JSL to this memory-protection address range, the code inside this range is executed in real time. When the matching RTL is encountered, execution returns to single-step or trace mode. While the code inside a code-trace address range is being executed, the line in the Memory-Protection subdisplay specifying that range is highlighted.
W	Code-window range. If one or more code-window address ranges are specified, code is executed only if it is inside one of the code-window ranges. In trace or single-step mode, whenever the program counter (K/PC) equals an address not in any of the code-window address ranges, execution stops. (If you don't specify any code-window address ranges, code can be executed at any address.) While the code inside a code-window address range is being executed, the line in the Memory-Protection subdisplay specifying that range is highlighted.
H	Trace-history window. Press H while editing a line to set the line as a trace history window. The debugger only saves instructions that are inside of the trace window to history. If you do not set any windows, then the debugger saves all instructions to history.

Table 16.24: Memory-Protection Code Settings

The initial Master display configuration provides nine lines for memory-protection ranges. When you start GSBug, the first memory-protection line is set to E1/0000-000F T; this code-trace range runs Apple IIGS tool calls in real-time mode. You can delete breakpoint lines to increase the number of memory-protection lines, or you can delete memory-protection lines to

increase the number of breakpoint lines. See the section "Configuring the Master display" later in this chapter for more information on customizing the Master display.

To alter the contents of the Memory-Protection subdisplay, type `MP` on the command line and press Return.

The `Mp` command moves the cursor to the first line of the Memory-Protection subdisplay. You can now use any of the keypress commands shown in Table 16.25.

<u>Command</u>	<u>Action</u>
Return	Moves to the next address down (or if the last line, wraps to the top).
<code>down-arrow</code>	Moves to the next address down (or if the last line, wraps to the top).
<code>up-arrow</code>	Moves to the next address up.
<code>left-arrow</code>	Moves left to the starting address. Type in the starting address of the code-trace or code-window range. You do not have to type a slash (/) after the bank value. If you do not include the bank number, the current value of the K register is used for the bank.
<code>right-arrow</code>	Moves right to the ending address. Type in the ending address of the code-trace or code-window range. Do not include a bank value; the bank must be the same as that of the starting address.
<i>hexadecimal_digit</i>	Used to enter addresses.
H	Sets the line you are editing as a Trace History window (indicated by an H).
T	Sets this line as a code-trace range.
W	Sets this line as a code-window range.
Delete	Deletes the current memory-protection line and increases the number of breakpoint lines by one.
?	Displays a help screen. Press any key except Esc to return to the Memory-Protection subdisplay.
Esc	Returns to the command line.

Table 16.25: Key Press Commands in the Memory Protection Subdisplay

For example, to enter a new code-window range – from 01/1220 to 01/12E5 – on the second line of the Memory-Protection subdisplay, use the following sequence of commands:

<u>Command</u>	<u>Meaning</u>
1. <code>MP</code> Return	Begin editing the Memory-Protection subdisplay.
2. <code>down-arrow</code>	Move down to the second address.
3. 11220	Type in the starting address.
4. <code>right-arrow</code>	Move right to the ending address.
5. 12E5	Type in the ending address.
6. W	Set this line as a code-window range.
7. Esc	Return to the command line.

To increase the number of memory protection ranges, you delete breakpoints as described in "Breakpoint Subdisplay" earlier in this chapter.

Command Line Subdisplay

KEY	BRK	DebugD	K/PC	B	D	S	A	X	Y	M	Q	L	P	nvmdizc	e
00	o d	0100	12/102E E1	0800	01FD	1F00	3AB9	0007	5F	33	1	30	00110000	0	
01FF:23		00/0305: E1 'a'	E0/100D-03-01												
01FE:9D		01/08E0: 9DBF	12/1103-00-00												
01FD:66		01/2003: 121200	00/0000-00-00												
01FC:DA		02/0BEA: 40 '█'	00/0000-00-00												
01FB:A3		00/0000: A0 ' '	00/0000-00-00												
01FA:39		00/0000: A0 ' '	00/0000-00-00												
01F9:5A		00/0000: A0 ' '	00/0000-00-00												
01F8:5A		00/0000: A0 ' '	00/0000-00-00												
01F7:5A		00/0000: A0 ' '	00/0000-00-00												
01F6:5A		00/0000: A0 ' '													
01F5:A9		00/0000: A0 ' '													
01F4:39		00/0000: A0 ' '													
01F3:FF		00/0000: A0 ' '													
01F2:3F		00/0000: A0 ' '													
01F1:23		00/0000: A0 ' '													
01F0:00		00/0000: A0 ' '													
01EF:01		00/0000: A0 ' '													
01EE:A4		00/0000: A0 ' '													
01ED:3E		00/0000: A0 ' '													

Figure 16.14: Command Line Subdisplay Area

Many of GSBug's functions are executed by typing a command while in the Master display. Commands are shown on the command line as you type them. Press Return to execute the command.

Unless otherwise stated, all commands are case insensitive.

Entering Commands

You can edit commands as you type them in with the functions listed in Table 16.26.

<u>Keystroke</u>	<u>Action</u>
Control-E	Toggles between insert and replace modes. Insert mode puts new characters between characters on the line, pushing the remaining characters to the right to make room. Replace mode puts new characters over the characters that the cursor is on.
left-arrow	Moves the cursor one character to the left.
right-arrow	Moves the cursor one character to the right.
Control-D	Deletes the character the cursor is on.
Delete	Deletes the character to the left of the cursor.
Control-Y	Deletes from the cursor position to the end of the line.
Control-X	Deletes the entire line.
Return	Executes the command that you typed on the command line and sends the entire line to the command interpreter; it does not truncate the line at the cursor position.

Table 16.26: Command Line Editing Functions

Viewing Memory with Templates

You can view memory through templates to display data structures in their proper format. Three commands let you use templates (see Table 16.27).

<u>Command</u>	<u>Action</u>
Loadtemp <i>pathname</i>	Loads the template file specified by <i>pathname</i> .
Killtemp	Unloads all templates currently in memory.
<i>_name address</i>	Views the memory starting at <i>address</i> through the template specified by <i>name</i> .

Table 16.27: Commands for Viewing Memory with Templates

If the debugger detects an error while loading a template file, it displays the error number on the command line. An error \$FFFF occurs when the template file is not in the correct format.

A template file consists of one or more template definitions. Each template definition must start with the line

Start *template_name*

where *template_name* is the name of the template.

Each template definition must end with the line

_End

The debugger disregards any information that is not placed between an _Start and an _End and treats this information as comments.

Each line within a template definition corresponds to one or more lines in the template display and includes three fields: label, type, and count. Each line must follow this syntax, with one or more spaces or tabs separating the different fields:

label *type* *count*

The label field is optional and contains a label that will appear before the information represented by the line. This label can consist of any combination of characters but cannot contain any spaces or tabs. The label field must be left justified (that is, have no spaces or tabs preceding it.) If you do not include the label field in the template definition, no label will appear before the data represented by the line. If the label field is left out, you must indent the type field.

The type field is optional and indicates the type of the data represented by the line:

Shell Reference Manual

<u>Type</u>	<u>Meaning</u>
byte	A hexadecimal byte.
word	A hexadecimal 16-bit word.
long	A hexadecimal 32-bit word.
ascii	An ASCII character.
cstring	An ASCII string terminated by a \$00.
pstring	An ASCII string preceded by a count byte.

Table 16.28: Type Field of Templates

You can abbreviate any of the types in a template field to its first letter. If you do not include the type field, the debugger does not print any data after the label. If you omit the type field, the debugger does not allow a count field.

If you omit both the label and type fields (in other words, there is a blank line in the template definition), a blank line appears in the template display.

The count field is optional and should be a decimal integer between 1 and 65,535. This number tells the debugger how many instances of the particular type of data to display. If you do not include the count field in your template definition, one instance appears.

Here is a typical template file:

```
_START example
one_byte      byte
some_words    word      3
some_longs    long      5
some_chars    ascii     26
a_C_string    cstring
a_P_string    pstring
_END

_START CREATE
pathname      l
access        w
file_type     w
aux_type      l
storage_type  w
create_date   w
create_time
_END
```

The file above contains two templates: example and CREATE. The example template is a sample template illustrating all the different data types, while CREATE is a template for the parameter list of a GS/OS Create call.

You could use the example template enter the command

```
_example 01/2000
```

If the data stored at \$01/2000 were

```

$01/2000: 5F 23 01 67 45 AB 89 67 45 23 01 EF CD AB 89 9A
$01/2010: AF 92 5D 7C BB 95 D2 20 A3 3F A8 61 62 63 64 65
$01/2020: 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75
$01/2030: 76 77 78 79 7A 48 65 6C 6C 6F 20 77 6F 72 6C 64
$01/2040: 21 00 06 48 6F 77 64 79 2E

```

the output would look like

```

one_byte      $5F
some_words    $0123 $4567 $89AB
some longs    $01234567 $89ABCDEF $5D92AF9A $D295BB7C
              $A83FA320
some_chars    abcdefghijklmnopqrstuvwxyz
a_C_string    Hello world!
a_P_string    Howdy.

```

You can load multiple template files into memory at one time.

The `Killtemp` command removes all currently loaded templates from memory.

Altering the Contents of Memory

To alter the contents of a memory location, whether displayed in the Memory subdisplay or not, you use the following command to set the contiguous memory starting at address to the values you specify:

address:value1 ...

The values you specify should be in one of the forms listed in Table 16.29.

<u>Form</u>	<u>Meaning</u>
<i>hex_byte</i>	A hexadecimal byte.
<i>short_address</i>	2-byte address stored with least significant byte first.
<i>long_address</i>	3-byte address stored with least significant byte first.
<i>" string "</i>	An ASCII string with the most significant bit of each byte set.
<i>' string '</i>	An ASCII string with the most significant bit of each byte cleared.

Table 16.29: Forms of Values for Modifying Memory

Do not precede hexadecimal bytes and addresses with a dollar sign and do not place a slash after the bank byte of long addresses.

For example, to place the hexadecimal byte `hex_byte` in memory starting at `address`, you would use the command

address:hex_byte

To enter a value of more than one byte, separate the values with spaces and enter the value that goes in the lowest address first. For example, to place the value `$A0` at `01/04ED` and the value `$A1` at `01/04EE`, you can use any of the following commands:

Shell Reference Manual

```
104ED:A0 A1
0104ED:A0 A1
1/04ED:A0 A1
01/04ED:A0 A1
```

In addition, if the K register is already set to 01, you can use the command

```
04ED:A0 A1
```

You can place values corresponding to strings ending with either string delimiter, " or ', no matter which one they began with. To enter string delimiters into memory, you use an exclamation mark (!). An exclamation mark in a string instructs the debugger to enter the next character into memory.

To enter a double quote into memory, type !" inside a string. To enter a single quote, type '!'. To enter an exclamation mark, type !!. For example, the following command:

```
04/2000:"!"It!'s true!!" she exclaimed."
```

enters the following string into memory at \$2000 in bank four.

```
"It's true!" she exclaimed.
```

When you need to enter many values into a line, press the Space bar. The next address appears on the command line followed by a colon (:).

If memory is modified while you are viewing it in a Memory display, the change appears immediately on the screen.

<u>Command</u>	<u>Action</u>
<i>address:" string</i>	Places values corresponding to <i>string</i> , with the high bit of each byte set, in memory starting at address. For example, the following command places the value \$E1 at 01/04ED and the value \$C1 at 01/04EE: 104ED:"aA To include in a string one of the characters used in commands, precede the character with an exclamation mark (!). For example, to put the string a"A into memory at 0104ED, placing the value \$E1 at 01/04ED, the value \$A2 at 01/04EE, and the value \$C1 at 01/04EF, use the command: 104ED:"a!"A
<i>address:' string</i>	Places values corresponding to <i>string</i> with the high bit of each byte cleared in memory at address. For example, the following command places the value \$61 at 01/04ED and the value \$41 at 01/04EE: 104ED:'aA

To include in a string one of the characters used in commands, precede the character with an exclamation mark (!). For example, to put the string a"A into memory at 0104ED, placing the value \$61 at 01/04ED, the value \$22 at 01/04EE, and the value \$41 at 01/04EF, use the command:

```
104ED:'a!'A
```

address:instruction

Assembles instruction and places the opcode and operand in memory starting at address. Simultaneously places the instruction on the last line of the Disassembly subdisplay. For example, the following command places the value \$A0 (the LDY immediate-address opcode) at 01/04ED and the value \$A1 at 01/04EE:

```
104ED:LDY #A1
```

You enter accumulator-mode expressions like implied-mode expressions: For example, you would enter ROL rather than ROL A. Branch instructions take the absolute address to branch to, not an offset. If you type:

```
LDA #FF
```

1. LDA #FF will appear in the Disassembly subdisplay regardless of the settings for e, m, and x.
2. When you execute that statement, it will change depending on e, m, and x.

You can combine values and strings in one command. To do so, separate values with spaces and include trailing delimiters for strings (that is, if the string begins with a single quotation mark ('), end it with a single quotation mark; if it begins with a double quotation mark ("), end it with a double quotation mark). For example, the following command places the values \$A0 \$A1 \$C1 \$F0 \$F0 \$EC \$E5 \$20 \$49 \$49 in memory starting at address 01/04ED:

```
01/04ED:A0 A1 "Apple"' II'
```

Table 16.30: Commands for Altering the Contents of a Memory Location

Hexadecimal-Decimal Conversions

GSBug can convert hexadecimal numbers to decimal and vice versa. To convert a number, type one of the commands in Table 16.31 on the command line and press Return.

<u>Command</u>	<u>Action</u>
<i>value</i> =	Converts <i>value</i> from hexadecimal to decimal. This command is identical to the \$ <i>value</i> command.
\$ <i>value</i> =	Converts <i>value</i> from hexadecimal to decimal. This command is identical to the <i>value</i> command.
+ <i>value</i> =	Converts <i>value</i> from decimal to hexadecimal.
- <i>value</i> =	Converts <i>value</i> from decimal to hexadecimal. A negative decimal value is converted to a 2-byte two's complement hexadecimal equivalent; for example, -10 = \$FFF6. (Note that if you put in \$FFF6, you get 65526, not -10.)

Table 16.31: Commands for Hexadecimal-Decimal Conversion

Evaluation of Expressions

GSBug evaluates expressions with the command

expression=

Table 16.32 lists the operators the debugger recognizes in expressions.

<u>Operator</u>	<u>Meaning</u>
+	Addition
-	Subtraction
*	Multiplication
/	Integer division
%	Modulus

Table 16.32: Operators Recognized in Expressions

The order of evaluation is from left to right, and all operators have equal precedence.

You can enter values in the expression as hexadecimal numbers, (preceded by a dollar sign or not as you wish) or as decimal numbers, which must be preceded by a positive or negative sign. For example, to add a decimal 6 to a decimal 5, you would enter the following expression:

+6++5=

Values can be a maximum of four bytes long. The debugger truncates longer values to four bytes. It converts negative decimal numbers to 4-byte two's-complement hexadecimal.

The bases and number of the values in the expression determine the base of the result. If the expression has at least two values, and all the values are in decimal, the result is also in decimal. If there are at least two values and one or more of them is in hexadecimal, the result is also in hexadecimal.

Decimal results of expressions are preceded by a plus (+).

If the expression consists of only one value, the result is in the opposite base (that is, hexadecimal if the value is in decimal and decimal if the value is in hexadecimal). You can use this fact to convert between bases.

The result of an expression appears on the command line and does not disappear with the next keystroke, so the result can be used in a further calculation or converted to the opposite base. For example, if you typed the command

```
13+1F=
```

the result would appear on the command line as follows:

```
1B+1F=$00000032
```

If you typed an equal sign and pressed Return, the command line would display

```
1B+1F=$00000032=+0000000058
```

To clear the command line, press Esc.

Configuring the Master Display

You can configure the GSBug Master display to meet your needs by adjusting the relative position of the stack pointer in the Stack subdisplay, the position of the current line in the Disassembly subdisplay, and the numbers of memory-protection lines and breakpoint lines. You can also select the slot used to send information to the printer.

To set the printer slot and adjust the positions of the stack pointer and current-instruction line, type `set` on the command line and press Return.

The following prompt appears on the command line:

```
dump alignment = ON                printer slot = 1
```

Type any number from 1 to 7 to set the slot that the debugger will use to send data to the printer. Type `s` to toggle between absolute and relative stack addresses. Type `d` to toggle memory dump alignment on and off.

You can also use the arrow keys to adjust the display. When you are done, press Esc to enter the changes and clear the command line. The actions of the arrow keys are shown in Table 16.33.

<u>Command</u>	<u>Action</u>
<code>left-arrow</code>	Moves the stack pointer up one line. All the stack subdisplay lines move up one line so that the highlighted line that indicates the current position of the stack pointer is one line higher in the display. You can now see the contents of one additional byte on the stack below (that is, with a lower address than) the stack pointer, and of one less byte above the stack pointer.
<code>right-arrow</code>	Moves the stack pointer down one line. All of the stack subdisplay lines move down one line, so that the highlighted line that indicates the current position of the stack pointer is one line lower in the display. You can now see the contents of one additional byte on the stack above the stack pointer, and of one less byte below the stack pointer.
<code>up-arrow</code>	Moves the current instruction up one line. When you type <code>set</code> and press Return, any disassembled code on the screen is cleared and an inverse-video bar appears at the location at which the current instruction would appear in

	the display. Each time you press the Up Arrow key, the highlighted bar moves up one line. You can now see the disassembly of one additional instruction following the current instruction, and of one less instruction preceding the current instruction when you continue single-stepping or tracing code.
<code>down-arrow</code>	Moves the current instruction down one line. Each time you press the Down Arrow key, the highlighted bar moves down one line. You can now see the disassembly of one less instruction following the current instruction and of one additional instruction preceding the current instruction when you continue single-stepping or tracing code.
<code>S</code>	Toggles between absolute and relative stack addresses.
<code>D</code>	Toggles memory dump alignment on and off.
<i>Number</i>	Sets default printer slot number (from 0 to 7).

Table 16.33: Commands to Configure the Master Display

Press Esc to complete your changes and clear the command line.

The Breakpoint and Memory-Protection subdisplays can each occupy from 1 to 17 lines in the Master display, but the total of both displays is always 18 lines. In other words, to increase the number of breakpoint lines, you must delete a corresponding number of memory-protection lines, and vice versa. To enter the Memory-Protection subdisplay, type `MP` on the command line and press Return. Then to delete a memory-protection line, use the arrow keys to highlight the line you want to eliminate, and press Delete. You can delete as many lines as you wish, except that at least one memory-protection line must remain on the screen.

To delete a breakpoint line, type `bp` on the command line and press Return to enter the Breakpoint subdisplay. Then use the arrow keys to highlight the line you want to eliminate, and press Delete. You can delete as many lines as you wish, except that at least one breakpoint line must remain on the screen.

If you edit the Breakpoint subdisplay when real breakpoints are in, the debugger automatically takes them out when you enter the subdisplay and puts them back in when you exit.

Saving a Display Configuration

Once you have customized the GSBug display to suit your needs, you can save the configuration to disk in a display-configuration file. Table 16.34 lists the information saved in a display-configuration file with associated commands.

<u>Type of Information Saved</u>	<u>Associated Command</u>
The position of the stack pointer in the Stack subdisplay.	Set
The position of the current instruction in the Disassembly subdisplay.	Set
The slot number for the printer.	Set
The stack address format.	Set
The dump alignment format.	Set
The memory addresses to be displayed and the type of display for each (hexadecimal, short address, or long address).	Mem
The number of memory-protection lines.	Mp
The memory-protection ranges and the type of range for each (code trace or code window).	Mp
The number of breakpoint lines.	Bp
The breakpoints, including the address and trigger value.	Bp

Table 16.34: Display Configuration Information and Commands

You can save as many configurations as you wish. To save and restore display configurations, type the commands in Table 16.35 on the command line, and press Return:

<u>Command</u>	<u>Action</u>
<i>C</i> Save <i>pathname</i>	Saves the current display configuration on disk to the file specified by <i>pathname</i> . Include the prefix for the file if you want to save it to a subdirectory other than the current GS/OS system subdirectory. For example, to save the current configuration to the file CONFIG.STORE in the directory /PROGRAMS/DEBUG/, use the command CSAVE /PROGRAMS/DEBUG/CONFIG.STORE
<i>C</i> Load <i>pathname</i>	Restores a previously saved display configuration from the disk file specified by <i>pathname</i> . Include the prefix for the file you want to use if the path name is different from the current GS/OS system subdirectory.

Table 16.35: Commands for Saving and Restoring Display Configurations

The debugger loads a configuration file at startup time containing the information saved with the CSave command. This standard configuration file, GSBUG.SETUP, should be located in the System.Setup folder of the disk containing the debugger; otherwise, it will not load automatically (you can still load it manually).

Command Line Commands

Table 16.36 lists all the commands available from the Master display command line. Most of these commands are described in detail elsewhere in this chapter, but they are included here for your

Shell Reference Manual

convenience. You can include a slash (/) after the bank value or omit it when entering an address; either form works. If you do not include the bank number, the current value of the K register is used for the bank. Press Return after each command-line command.

<u>Command</u>	<u>Action</u>
<code>?</code>	Displays a help screen.
<code>address:</code>	Displays 368 contiguous bytes of memory starting at <i>address</i> .
<code>address::</code>	Displays memory starting at 2-byte address stored at <i>address</i> ; can be used for dereferencing a pointer (indirection).
<code>address:::</code>	Displays memory starting at 3-byte address stored at <i>address</i> ; can be used for dereferencing a pointer (indirection).
<code>address1.address2:</code>	Displays a range of memory from <i>address1</i> to <i>address2</i> .
<code>address:instruction</code>	Assembles instruction and places the opcode and operand in memory starting at address. Simultaneously places the instruction on the last line of the Disassembly subdisplay.
<code>address:'string'</code>	Places values corresponding to <i>string</i> , with the high bit of each byte cleared, in memory at <i>address</i> .
<code>address:"string"</code>	Place values corresponding to <i>string</i> , with the high bit of each byte set, in memory starting at <i>address</i> .
<code>address:value</code>	Places the hexadecimal value <i>value</i> in memory starting at <i>address</i> . To enter a value of more than one byte, enter the byte that goes in the highest address first.
<code>register=value</code>	Sets the register specified by <i>register</i> to the value specified by <i>value</i> . This command is case sensitive.
<code>value=</code>	Converts <i>value</i> from hexadecimal to decimal. This command is identical to the <code>\$value=</code> command.
<code>\$value=</code>	Converts <i>value</i> from hexadecimal to decimal. This command is identical to the <code>value=</code> command.
<code>+value=</code>	Converts <i>value</i> from decimal to hexadecimal.
<code>-value=</code>	Converts <i>value</i> from decimal to hexadecimal. A negative decimal value is converted to a 2-byte two's complement hexadecimal equivalent.
Space bar	Writes the next available address on the command line, followed by a colon. Use this command to get the next address after using any command starting with <i>address:</i> .
<code>Asm</code>	Clears the Disassembly subdisplay to prepare to enter a sequence of instructions using the <i>address:instruction</i> command.
<code>Cload pathname</code>	Restores a previously saved display configuration from the disk file specified by <i>pathname</i> .
<code>Clr</code>	Clears all breakpoints to 00/0000-00-00.
<code>CSave pathname</code>	Saves the current display configuration on disk to the file specified by <i>pathname</i> .
<code>DP:</code>	Displays the direct page.
<code>DPAGE</code>	Sets D (Direct-Page register) to the direct page the debugger allocated for the user at startup.
<code>Dbrk</code>	Returns to the debugger when a BRK instruction that has not been set as a breakpoint is encountered while your program is running in real-time mode.

<i>e</i>	Toggles the e flag: if it's set to 1, changes it to 0; if it's set to 0, changes it to 1. This command is case sensitive.
<i>expression=</i>	Evaluates <i>expression</i> (The order of evaluation is from left to right with all operators having equal precedence).
In	Inserts real breakpoints.
<i>addressG</i>	Jumps directly to code at address <i>address</i> . If you omit <i>address</i> , the current K/PC address is used.
<i>addressJ</i>	Jumps directly to code at address <i>address</i> . If you omit <i>address</i> , uses the current K/PC address.
<i>Key=keynum</i>	Each bit of the binary number represented by the hexadecimal number <i>keynum</i> specifies one key to be used as a keystroke modifier; sets that bit to 1 to make that key a keystroke modifier. The bit assignments are described in the section "Register Subdisplay" in this chapter.
Load <i>pathname</i>	Loads the program specified by <i>pathname</i> to debug.
<i>m</i>	Toggles the m flag: if it's set to 1, changes it to 0; if it's set to 0, changes it to 1. This command is case sensitive and works only if <i>e=0</i> .
Monrk	Exits to the Monitor when a BRK instruction that has not been set as a breakpoint is encountered while in real-time mode.
Mon	Exits from the debugger into the Monitor. Press Control-Y and Return to return to the debugger.
Off	Turns off the Master display and displays your program.
On	Turns off your program's display and turns on the Master display.
Out	Removes real breakpoints.
P	Prints the current text screen. You can use this command with the Master display on to print the current Master display, or with the Master display off to print your program's display (80-column text only). You can also print the Memory display or help screens with this command.
Prefix <i>n pathname</i>	Changes GS/OS prefix <i>n</i> to <i>pathname</i> . This command has the same effect as the shell's PREFIX command. If you omit <i>n</i> , prefix 0 is changed.
Q	Exits the debugger. This command terminates GSbug, unlike the Mon command, which allows you to return from the Monitor to the debugger. If you called the debugger from the shell, Q returns you to the shell. Q restores the state of the machine at exit.
QR	Exits the debugger. QR does not restore the state of the machine at exit; it leaves the system as modified.
<i>addressS</i>	Enters single-step mode at address. If you omit <i>address</i> , it uses the current setting of the K/PC register.
Set	Adjusts the positions of the stack pointer and current-instruction line and sets the printer slot.
Shutdown <i>UserID quit_flag</i>	Executes a UserShutDown System Loader call.
Stack <i>S</i>	Sets the address of <i>S</i> (<i>S=address</i>) of the user stack to the stack that was allocated by the debugger at startup.
<i>addressT</i>	Enters trace mode at address. If you omit <i>address</i> , uses the current setting of the K/PC register.

Tool # <i>toolnum</i>	Displays the name, number, and actual address of the tool number <i>toolnum</i> .
Tool <i>_toolname</i>	Displays the name, number, and actual address of the tool named <i>toolname</i> .
Unload	Unloads GSbug and then quits (not available in the application version).
V	Displays the current version number and copyright of GSbug.
x	Toggles the x flag: if it's set to 1, changes it to 0; if it's set to 0, change it to 1. This command is case sensitive and works only if e=0.
<i>addressx</i>	Executes a JSL (or real time JSR) directly to code at address. If you omit address, uses the current setting of the K/PC register. If you omit address, the X must be uppercase.

Table 16.36: Commands Available from Master Display Command Line

Part 4: Loader Dumper

Loader Dumper is a desk accessory that you can use together with GSbug to debug relocatable and dynamic code. Loader Dumper lets you see where in memory the System Loader has loaded each segment of your program and gives you information about the various tables and variables that the loader uses. The System Loader is described in the GS/OS Reference.

To get the Desk Accessories menu, press ⌘-Control-Esc. When you select Loader Dumper from the Desk Accessories menu, the menu shown in Figure 16.15 appears on the screen.

1. Dump Memory Segment Table
2. Dump Pathname Table
3. Dump Jump Table
4. Dump Loader Globals
5. Dump GS/OS Packets
6. Dump File Buffer Variables
7. Get Load Segment Information
8. Get UserID Information

What do you want to dump ?

Figure 16.15: Loader Dumper main menu

Type the number associated with the menu item of your choice.
All these selections are described in the following sections.

Dump Memory Segment Table

The memory segment table is a linked list, each entry of which describes a memory block known to the System Loader. Each memory block corresponds to a single load segment. Note that dynamic segments do not appear in the memory segment table when the program is initially loaded because they are not loaded into memory until the program needs them.

You can use the memory segment table to get the starting address of every segment currently in memory. The entries in this table are shown one at a time. Press Return to see the next entry. Press Esc to return to the Loader Dumper main menu. To see memory segment table information

on one specific segment (instead of scrolling through the entire memory segment table), use selection 7, Get Load Segment Information.

Before using the memory segment table to get the starting addresses of segments, you must know the user ID and file number of the program in which you are interested. This information is available from selection 2, Dump Pathname Table. If you load a program with GSbug, the user ID is also displayed in the A register immediately after the program is loaded.

The starting address of the segment appears in parentheses after the Memory Handle field. For example, the starting address in memory of load segment 1 of load file 1 for UserID \$A001 (as shown in Figure 16.16) is \$1188D3.

The type of segment is shown in parentheses after the Load Segment Kind field.

```
Memory Segment Table
$E11854 (1188BF)
-----
Next Handle      = $E118B8 (11FD8B)
Prev Handle      = $000000
-----
UserID           = $A001
Memory Handle     = $E11840 (1188D3)
Load File Number  = $0001
Load Segment Number = $0001
Load Segment Kind = $2000 (Position Independent)

Press RETURN to continue

$E118B8 (11FD8B)
-----
Next Handle      = $E118E0 (11FD6F)
Prev Handle      = $E11854 (1188BF)
-----
UserID           = $5002
Memory Handle     = $E118A4 (020000)
Load File Number  = $0001
Load Segment Number = $0001
Load Segment Kind = $0402 (Jump Table Segment)
                  (Reload)

Press RETURN to continue
```

Figure 16.16: Memory Segment Table Output

If your program has unloaded a memory block (that is, made it purgeable), you can use the memory segment table to find out if it has been purged. To do so, check the address in parentheses after the memory handle: If the address is 000000, the block has been purged.

Dump Path Name Table

The path name table provides a cross-reference between file numbers, file path names, and user IDs. The path name table is a linked list of individual path name entries. The entries in the table are shown one at a time; press Return to see the next entry. Press Esc to return to the Loader Dumper main menu.

You can use the path name table to get the user ID and file number of every program in memory. You need this information to use the memory segment table to find the starting memory address of a segment. The path name table also gives you the starting address and size of the direct-page/stack space requested by the loader for each program. (The loader requests a direct-page/stack space only if you include a direct-page/stack segment in your program; otherwise GS/OS either assigns the direct-page/stack space as a default or your program can request one through the Memory Manager.)

The path name table display of the Loader Dumper is illustrated in Figure 16.17.

```

Pathname Table
$E11890 (118527)
-----
Next Handle      = $E118F4 (11F9F6)
Prev Handle      = $000000
-----
UserID           = $A001
File Number      = $0001
File Date        = $06000617570A0B00 (7/24/87 10:11:0)
Direct Page/Stack Addr = $0000
Direct Page/Stack Size = $0000
Jump Table Segment Flag = $0001
Starting Address  = $00060000
File Pathname     = /ORCA/SYSTEM/SYSTEM.SETUP/TOOL.SETUP

Press RETURN to continue

$E118F4 (11F9F6)
-----
Next Handle      = $E11930 (11FAD9)
Prev Handle      = $E11890 (118527)
-----
UserID           = $5002
File Number      = $0001
File Date        = $AD2C
Direct Page/Stack Addr = $0000
Direct Page/Stack Size = $0000
File Pathname     = /ORCA/SYSTEM/DESK.ACCS/MANGLER.DA

Press RETURN to continue

```

Figure 16.17: Path Name Table Output

Dump Jump Table

All references to dynamic segments are made through the jump table. The jump table in memory consists of the jump table directory and one or more jump table segments. The jump table directory is a linked list, each entry of which points to a single jump table segment encountered by the loader. The Loader Dumper displays each jump table directory entry followed by the jump table segment to which the entry points. Each jump table segment contains one entry for each reference to a dynamic segment in the program.

The entries in the table are shown one at a time; press Return to see the next entry. Press Esc to jump to the next directory entry. If you are at the last directory entry, Esc returns you to the Loader Dumper main menu.

You can use the jump table to determine whether a dynamic segment has been loaded into memory; if it has been loaded, you can use the memory segment table to find the starting address of the segment in memory. A sample Jump table display is shown in Figure 16.18. The first entry in Figure 16.18 is for a dynamic segment that has been loaded into memory. You can tell that the segment has been loaded because the jump table segment entry is in its loaded state; it ends in a JML to the referenced subroutine. The operand of the JML statement is the location in memory of the subroutine being referenced (if there is more than one routine or entry point in the segment, there will be more than one jump table entry for that segment). The number in parentheses after the Handle to Segment field shows the location in memory of the jump table segment itself.

The second entry in Figure 16.18 is for a dynamic segment that has not been loaded into memory. The jump table segment entry ends in a JSL to the System Loader's Jump table load function.

Before using the jump table to get information about dynamic segments, you must know the user ID and file number of the program in which you are interested. This information is available from selection 2, Dump Pathname Table.

The jump table and jump table segments are described in the GS/OS Reference.

Jump Table

```
$E11AC0 (11FA2D)
-----
Next Handle      = $E11818 (11E2AA)
Prev Handle      = $E11840 (11FA3B)
-----
UserID          = $1007
Handle to Segment      = $E11A34 (010A6B)

UserID          = $1007
Load File Number      = $0001
Load Segment Number   = $0002
Load Segment Offset   = $00000000
Jump to Loader/Function = JML 010A85
```

Press RETURN to continue

```
$E11818 (11E2AA)
-----
Next Handle      = $000000
Prev Handle      = $E11AC0 (11FA2D)
-----
UserID          = $1008
Handle to Segment      = $E118E0 (010B13)
```

Shell Reference Manual

```
UserID      = $1008
Load File Number      = $0001
Load Segment Number   = $0002
Load Segment Offset   = $00000000
Jump to Loader/Function = JSL 11FF10
```

Press RETURN to continue

Figure 16.18: Jump Table Output

Dump Loader Globals

The Loader globals display shows the values of some loader variables and some statistics associated with the last load operation performed. This table was included primarily for use by Apple engineers when they were debugging the System Loader. If you have trouble using the loader, or believe you have found a bug in the loader, copy down the information in the Last Function, Total Errors, and Error Addresses fields before calling technical support, or include this information in your bug report. Press Return or Esc to return to the Loader Dumper menu.

```
Loader Globals
-----
$01FB00 SEGTLB      = $E11A20 (11FDC8)
$01FB04 JMPTBL      = $000000
$01E70A PATHTLB     = $E117A0 (11FDAC)
$01E70E USERID      = $1002
$01E72C Last Function = $0022 (Get Pathname 2)
$01E710 Total Errors = $0000
$01FB10 LastError    = $0000
$01FB12 Error_Addresses = $F71C F6ED E6A3 B024
$01FB1A LCJumpLoad   = $11FF10
$01FB1E LCReturn     = $11FF36
$01E72E nLCONST      = $0012
$01E730 nRELOC        = $0000
$01E732 nINTERSEG    = $0000
$01E734 nDS          = $0011
$01E736 ncRELOC       = $1262
$01E738 ncINTERSEG    = $0000
$01FB30 nSUPER        = $0002
$01E73A nSegments     = $0001
$01E73C nBytes        = $0000A1CD
```

Press RETURN to continue

Figure 16.19: Loader Globals Output

The fields beginning with a lowercase n (nLCONST or nRELOC – for an example, see Figure 16.19) show the number of certain kinds of records, the number of segments, and the number of bytes loaded by the last Initial Load call.

Dump GS/OS Packets

This display shows the GS/OS calls, including parameter blocks, used most recently by the loader. This information is primarily for use by Apple engineers in debugging the loader and GS/OS. The GS/OS Packets display is illustrated in Figure 16.20. GS/OS calls are described in the GS/OS Reference.

```

GS/OS Packets
-----
$01FB8A POpen          = $30F8 0003 000FE0EB 4000 4000 4000 0E72 8EBC0E72 0000
                        = $0001000A0009D2E (1/11/0 0:57:46)
                        = $00000010011FF6A (1/2/0 17:55:06)
$01FBB0 PRead          = $00E3 000000B3 00020000 58103200
$01FBBE PClose         = $0B0B
$01FBC0 PGetMark       = $0200 58103300
$01FBC6 PSetMark       = $0B0B 00010200
$01FBCC PExpandPath    = $E0EB 4000000F 0E720000 0000
$01FBD8 Psys_prefs     = $0000 0001

```

Press RETURN to continue

Figure 16.20: GS/OS Packets Output

Dump File Buffer Variables

The file buffer is used by the System Loader to buffer data being loaded from disk into memory. You can use the File buffer variables display to monitor the progress of a load. Press Return or Esc to return to the Loader Dumper main menu.

```

File Buffer Variables
-----
$01E79E File_Buff      = 00114527
$01E7A2 File_Buff_Size = 4000
$01E7A8 File_Pt        = 236B
$01E7AA File_EOB       = 236B
$01E7AC File_Mark      = 00010000
$01E7B0 Header_Mark    = 00012400

```

Press RETURN to continue

Figure 16.21: File buffer variables output

The File_Pt field shows the next location to be read from the file buffer.

The File_EOB field shows the location of the last valid data currently in the buffer. These fields are equal after a load is complete and anytime during a load that the buffer is full, indicating that all the data in the buffer has been read. In this case, the next read operation on the buffer will cause the buffer to be refreshed.

The File_Mark field shows the last location read from within the GS/OS file being loaded. The Header_Mark field shows the location of the beginning of the next segment header in the file being loaded.

Get Load Segment Information

The Get Load Segment Information selection provides the same information as the Dump Memory Segment Table selection, except that information is provided on only the segment you specify. Figure 16.22 illustrates the Get Load Segment Information selection; the characters shown in boldface are the ones you type in.

```

Key in UserID of Load Segment      - 5002
Key in File Number of Load Segment - 1
Key in Segment Number of Load Segment - 1
-----
UserID                            = $5002
Memory Handle                     = $E118A4 (020000)
Load File Number                  = $0001
Load Segment Number               = $0001
Load Segment Kind                 = $0000
-----

Press RETURN to continue

Key in UserID of Load Segment -

```

Figure 16.22: Load segment information output

Before using this selection to get information about static segments, you must know the user ID and file number of the program in which you are interested. This information is available from selection 2, Dump Pathname Table.

Press Return to be prompted for the next load segment. Press Esc to return to the Loader Dumper main menu.

Get User ID Information

The Get userID information selection lets you display all the load segments in an application. Figure 16.23 illustrates the Get UserID Information selection; the characters shown in boldface are the ones you type in.

Before using this selection to get information about load segments, you must know the user ID and file number of the program in which you are interested. This information is available from selection 2, Dump Path Name Table.

```

Key in UserID - 1005
-----
/GSBUG/DEBUG
File Num = $0001 Segment Num = $0001 Kind = $0001 Address = $010800-01A41F
-----
Key in UserID -

```

Figure 16.23: Get User ID Information Output

The File Num field displays the number of the initial load file. Other files numbers are usually run time libraries associated with the initial load file.

The Segment Num field shows the segment number in the file that is being displayed.
 The Kind field displays the Kind attribute in the segment header.
 The Address field shows the actual range of memory where the segment is loaded.
 An application may display more than one file and more than one segment if a more complex library is loaded as shown in Figure 16.24.

```
Key in UserID - 1001
-----
/HD/TDIR/PROG
File Num = $0001  Segment Num = $0001  Kind = $0000  Address = $012C1C-012C84
File Num = $0001  Segment Num = $0002  Kind = $0002  Address = $012C85-012CC8
File Num = $0001  Segment Num = $0003  Kind = $0004  Address = $012CC9-012CF4

/HD/TDIR/RUNLIB1
File Num = $0002  Segment Num = $0002  Kind = $8000  Address = $012CF5-012D39
File Num = $0002  Segment Num = $0003  Kind = $8000  Address = $012D3A-012D7E

/HD/TDIR/RUNLIB2
File Num = $0003  Segment Num = $0002  Kind = $8000  Address = $012D7F-012DC3
File Num = $0003  Segment Num = $0003  Kind = $8000  Address = $012DC4-012E08
-----
Key in UserID - 1001
```

Figure 16.24: Multiple Segments in Get userID Information Output

Chapter 17

Running the Assembler

In this chapter, we will cover the assembly process. This material is the groundwork for the remaining chapters in the Assembler Reference Manual. Topics covered are:

- Invoking the assembler.
- The assembly process.
- Reading the assembly listing.

Introduction

The assembler is the heart of the ORCA/M assembly language development system. It is invoked from the shell 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.

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.

Shell Commands That Assemble A Program

There are several commands that accomplish program assembly. Depending on your application, there are commands that: assemble a program; assemble, then link your program; assemble, then link, then execute your program. These commands are summarized below.

ASSEMBLE	Assemble a program.
ASML	Assemble and link a program.
ASMLG	Assemble, link, and execute a program.
COMPILE	Compile a program; same as ASSEMBLE.
CMPL	Compile and link a program; same as ASML.

CMPLG Compile, link, and execute a program; same as ASMLG.

RUN Assemble (compile), link, and execute a program.

In the ORCA environment, issuing an assemble or compile command produces the same result: the shell interrogates the language stamp of the source file, and then invokes the language translator defined for the source file's stamp.

Assembler Command Options

The assemble commands have several command line parameters that can modify the results of an assembly. These parameters allow: the handling of errors during assembly, the display or listing of assembly results, specifying the output file name from the command line, and specification of the segments you want to assemble during a partial assembly. Default values for the parameters are underlined. The parameters which handle errors are described in Chapter 12.

The parameters L, S, and KEEP have corresponding assembler directives. If both a command line parameter and a directive are used, the command line parameter setting is used.

The formats for the commands showing what parameters are available are listed below. Detailed descriptions of the parameters follow.

ASSEMBLE	[+D -D] [+E -E] [+M -M] [+L <u>-L</u>] [<u>+P</u> -P] [+S <u>-S</u>] [+T <u>-T</u>] [+W <u>-W</u>] <i>sourcefile 1 sourcefile2 ...</i> [KEEP= <i>outfile</i>] [NAMES=(<i>seg1</i> [<i>seg2</i> [...]])]
ASML	[+D -D] [+E -E] [+M -M] [+L <u>-L</u>] [<u>+P</u> -P] [+S <u>-S</u>] [+T <u>-T</u>] [+W <u>-W</u>] <i>sourcefile 1 sourcefile2 ...</i> [KEEP= <i>outfile</i>] [NAMES=(<i>seg1</i> [<i>seg2</i> [...]])]
ASMLG	[+D -D] [+E -E] [+M -M] [+L <u>-L</u>] [<u>+P</u> -P] [+S <u>-S</u>] [+T <u>-T</u>] [+W <u>-W</u>] <i>sourcefile 1 sourcefile2 ...</i> [KEEP= <i>outfile</i>] [NAMES=(<i>seg1</i> [<i>seg2</i> [...]])]
COMPILE	[+D -D] [+E -E] [+M -M] [+L <u>-L</u>] [<u>+P</u> -P] [+S <u>-S</u>] [+T <u>-T</u>] [+W <u>-W</u>] <i>sourcefile 1 sourcefile2 ...</i> [KEEP= <i>outfile</i>] [NAMES=(<i>seg1</i> [<i>seg2</i> [...]])]
CMPL	[+D -D] [+E -E] [+M -M] [+L <u>-L</u>] [<u>+P</u> -P] [+S <u>-S</u>] [+T <u>-T</u>] [+W <u>-W</u>] <i>sourcefile 1 sourcefile2 ...</i> [KEEP= <i>outfile</i>] [NAMES=(<i>seg1</i> [<i>seg2</i> [...]])]
CMPLG	[+D -D] [+E -E] [+M -M] [+L <u>-L</u>] [<u>+P</u> -P] [+S <u>-S</u>] [+T <u>-T</u>] [+W <u>-W</u>] <i>sourcefile 1 sourcefile2 ...</i> [KEEP= <i>outfile</i>] [NAMES=(<i>seg1</i> [<i>seg2</i> [...]])]
RUN	[+D -D] [+E -E] [+M -M] [+L <u>-L</u>] [<u>+P</u> -P] [+S <u>-S</u>] [+T <u>-T</u>] [+W <u>-W</u>] <i>sourcefile 1 sourcefile2 ...</i> [KEEP= <i>outfile</i>] [NAMES=(<i>seg1</i> [<i>seg2</i> [...]])]

- +D|-D** The +D and -D flags are used to turn generation of debug code on and off. The assembler does not generate debug code, and ignores this flag.

- +E|-E** When a terminal error is encountered during an assembly from the command line, the system will normally stop execution, enter the editor, place the cursor over the offending line, and display the error message. In assemblies from an EXEC file, the system will simply stop execution when an error is encountered, writing the error message to error out. Using +E tells the assembler to enter the editor when a terminal error is found, while using -E tells the assembler not to enter the editor when an error is found.

- +L|-L** If you specify +L, the assembler generates a source listing, and if you are using LINK, a link map of the segments in the object file (including the starting address, the length in hexadecimal of each segment, and the segment type). If you specify -L, the assembler does not produce a listing, and the linker does not produce a link map. The L parameter overrides the LIST directive in the source file.

- +M|-M** +M causes any object modules produced by the assembler to be written to memory, rather than to disk. Do not use the +M flag if you will be performing partial assemblies or using separate compilation to build your final program.`

- +P|-P** The assembler normally writes progress information as the assembly progresses, writing the name of the various source segments as they are assembled. The linker also writes progress information, printing a dot as each segment is processed during pass one and pass 2 of the link. The -P flag suppresses all progress information.

- +S|-S** If you specify +S, the assembler produces an alphabetical listing of all local symbols following each END directive. The linker, if it has been called, produces an alphabetical listing of all global references in the object module, called a symbol table. If you specify -S, these symbol tables are not produced. The S parameter in this command overrides the SYMBOL directive in the source file.

- +T|-T** The +T flag tells the assembler to treat all errors as terminal errors, stopping the assembly as soon as the first error is encountered. The +T flag is most often used in conjunction with the +E flag, causing the assembler to start the editor, displaying the location and error message as soon as any error is found.

- +W|-W** When the +W flag is used, the assembler will stop and wait for a keypress after printing an error message. By default, the assembler does not stop after flagging an error.

sourcefile1 sourcefile2 The full or partial path names (including the file name) of the source files. Note that more than one source file can be included, and that object files and library files can also be mixed with the source files. The object files

produced by assembling each source file, along with any object or library files, are passed on to the linker, if no errors occurred during the assemblies.

Keep=*outfile* Use this parameter to specify the path name or partial path name, including the file name, of the output file, or the executable load file, if you are using LINK. If you are assembling a one-segment program, the output module is named *outfile*.ROOT. If the program contains more than one segment, the first segment is placed in *outfile*.ROOT and the other segments in *outfile*.A. If you are performing a partial assembly, you will see other file name extensions, described later in this manual.

This parameter has the same effect as placing a KEEP directive in your source file. If you have a KEEP directive in the source file and you also use the KEEP parameter, the KEEP directive takes precedence. In this case, two object modules are produced with the extension .ROOT; one corresponding to the parameter and one corresponding to the directive. However, the path name in the KEEP directive takes precedence; other files with .A or other extensions are created only with the file name used in the directive, and the linker uses only the path name given in the KEEP directive.

You should remember the following points in regard to the KEEP parameter:

- If you use neither the KEEP parameter nor the KEEP directive, then the shell can add a keep name automatically. This automatic naming is due to the shell variable {KeepName}. If you specify a name in the {KeepName} shell variable before an assembly is attempted, and you don't explicitly name the output file by the KEEP directive or KEEP command line parameter, then the keep name will be the path of the source file name, with the keep name appended. A \$ character in the keep name will expand to the source file name, with the last extension removed. A % character expands to the source file name. It is recommended that you specify the keep name explicitly in the KEEP directive or KEEP parameter. If you have not specified the KeepName variable (a null string) nor explicitly named the output file, then the output generated from the file is not kept. The examples below show the keep names that would be used with several KeepName settings, assuming the source file is called MY.FILE.A.

<u>{KeepName}</u>	<u>object file name</u>
\$.O	MYFILE.O
%	MYFILE.A
OUT.O	OUT.O

- The file name you specify as *outfile* must not be over 10 characters long. This is because the extension .ROOT is appended to the name, and GS/OS does not allow file names of over 15 characters.
- If you do not use the keep parameter, the keep directive, or the {KeepName} variable, no output file is produced.

NAMES=(*seg1 seg2 ...*) This parameter causes the assembler to perform a partial assembly. The names *seg1*, *seg2*, ... specify the source file segments to be assembled, rather than the entire file. Separate the names with spaces. The ORCA linker automatically selects the latest version of each segment when the program is linked.

You assign names to object segments with PRIVATE, PRIVDATA, START, and DATA directives. The object file created when you use the NAMES parameter contains only the specified object segments. When you link a program, the linker scans all the files whose file names are identical except for their extensions, and takes the latest version of each segment. Therefore, you must use the same output file name for every partial assembly of a program. No blanks are permitted immediately before or after the equal sign in this parameter.

Assembler Directives Global In Scope

In a partial assembly, certain directives are assembled whether or not they appear in a segment that is being assembled. This is because these directives can have side effects that outlast the segment they appear in. These directives are:

ABSADDR	Generate absolute addresses
APPEND	Append a source file
CASE	Specify case-sensitive
CODECHK	Disable code bank checks
COPY	Copy a source file
DATACHK	Disable data bank checks
DIRECT	Set a fixed direct-page value
ERR	Print errors
EXPAND	Expand DC Statements
GEN	Generate macro expansions
GEQU	Define global constant
IEEE	Enable IEEE format numbers
INSTIME	Generate instruction times
KEEP	Keep output file
LIST	List output
LONGA	Select accumulator size
LONGI	Select index register size
MCOPY	Copy macro file to macro buffer
MDROP	Drop macro file from macro buffer
MEM	Reserve memory
MERR	Set maximum error level
MLOAD	Load macro file into macro buffer
MSB	Set most significant bit
NUMSEX	Define byte order for floating-point numbers
OBJCASE	Specify case sensitive in object files
PRINTER	Send output to the printer
RENAME	Rename op code
SETCOM	Set comment column
SYMBOL	Print symbol tables

Assembler Reference Manual

TITLE	Print heading at top of each page of assembly listing
TRACE	Trace macros
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 chapters 19 and 20.

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, 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, PRIVATE, PRIVDATA, 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 direct-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 if +S has been coded on the command line, or SYMBOL ON appeared in the source file. 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 shell. Depending on the "assemble" command given, the shell passes control to either the command processor 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 relocatable file as output.

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 list the subroutine name).

To restart the listing, any key but (C.) may be pressed. The listing will continue until another key is pressed to stop it again. If the listing has been stopped, and C. 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. C. can also be used to directly abort the assembly, without first stopping the listing.

The assembler can stop automatically when it finds an error. To do this, use the +W (wait) flag. Restarting is accomplished by pressing any key. Once again, C. will abort to the editor, where the error can be fixed at once.

Terminal Errors

If the assembler encounters a terminal error (such as a symbol table overflow), it returns control to the shell. The shell then calls 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.

See the description of the ASML command in Chapter 12 for flags that let you make all errors terminal or prevent entry into the editor after a terminal error.

A list of terminal errors is contained in Appendix A.

The Assembly Listing

Screen Listings

A listing to standard output is produced during pass two if the assembler is instructed to list the output. If no listing is produced, each subroutine begins with two messages announcing the subroutine name and pass. These messages are not printed if the -P flag has been used to suppress progress information. If a listing is produced, the source code, along with some diagnostic information, is displayed. A sample listing for the "Hello world!" program given in Chapter 1 is shown in Figure 15.1.

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 direct page; the origin of \$0000 is simply for convenience in calculation.) Then 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.

The three lines at the very bottom of the listing give overall information about your program. In this case our program is 9 lines long, as indicated. We used three macros, (PUTS calls other macros), which is also noted. When the assembler expanded the macros, 28 additional lines were processed by the assembler. Because of the addition of 28 extra lines, the assembly for this program is a little slower than if only 9 lines were assembled.

If an error is detected in the source statement, an error message is printed on the next line. All error messages are text messages, not simply error numbers. The errors are explained in Appendix A. Several directives are provided which can modify the source listing. See Chapter 19 for details.

Assembler Reference Manual

ORCA/M Asm65816 2.0

```
0001 0000                                keep  Hello
0002 0000                                mcopy Hello.Macros
0003 0000                                Main  start
0004 0000 4B                            phk
0005 0001 AB                            plb
0006 0002                                puts  #'Hello, world.',cr=t
0007 001F A9 00 00                       lda   #0
0008 0022 6B                            rtl
0009 0023                                end

9 source lines
3 macros expanded
28 lines generated
```

Figure 15.1 An Assembly Listing Example

Printer Listings

If the **PRINTER ON** directive is issued in a source file, subsequent lines are sent to the printer. The assembler expects the printer to be on-line, and to be set up according to the instructions in Chapter 2. The listing can also be sent to the printer from the command line by using redirection:

```
#ASML +L MYPROG > .PRINTER
```

Printed listings are generally the same as listings to the screen. 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. These defaults can be modified by changing the shell variables that control printer options.

Chapter 18

Coding Instructions

This chapter covers ORCA assembly language program syntax. Assembly language programs have strict formats; failure to adhere to the rules covered in this chapter can cause your program to fail to assemble. The rules are few and easy to remember. The topics covered in this chapter are:

- Types of source statements.
- Comment lines.
- Instruction formats.

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 20.

Assembler source file lines may be up to 255 columns long, numbered from 1 to 255. 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. Note that many printers allow compression of text, so that more than eighty columns can be printed on a single line. If so, be sure and allow twenty-three columns for the assembler's output.

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 * ; !

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 20.) If you decide to use a line starting with a period for a comment, be sure and skip one space before starting the rest of the line.

Instructions

An assembly language statement, whether an instruction, directive or macro call, 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 embedded 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 as long as 255 characters. All characters are significant.

Note that labels starting with the tilde character are reserved for use in macros and libraries supplied by the Byte Works. To avoid conflicts with the standard libraries, you should avoid use of the tilde.

It is best not to use the single character 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 65816 operation codes:

<u>Standard</u>	<u>Also Allowed</u>
BCC	BLT
BCS	BGE
CMP	CPA

Assembler directives vary in length from two to eight characters. The operation codes for assembler directives are listed in Chapter 19 and Chapter 20. Macro operation codes, which are a form of user-defined operation code, are described in Chapter 20.

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 values are set by EQU directives or GEQU directives, and the EQU or GEQU directives have operands that are constant, 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 direct page addressing, 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 | for backward compatibility with versions of ORCA running on computers that do not have the | on the keyboard.

Operands for immediate addressing are resolved to two bytes; if you are using short registers then the operand will generate one byte. 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. For backward compatibility, the / character can be substituted for #>.

The following table shows all legal operands of both the 6502 and 65816. The labels DP, ABS and LONG refer to constant expressions that resolve to one-, two- or three-byte values, respectively. A relocatable expression may be used in place of DP, ABS, or LONG. The Code Generated field assumes native mode (16-bit registers). Assume that DP = \$01, ABS = \$0203, LONG = \$040506. Note that bytes generated are placed in memory in the format that the 65816 wants them: low byte, then high byte.

Assembler Reference Manual

<u>Addressing Mode</u>	<u>Operand Format</u>	<u>Operand Code Generated</u>
Implied	none needed	
Immediate	#DP	\$01 00
	#ABS	\$03 02
	#LONG	\$06 05
	#<DP	\$01 00
	#<ABS	\$03 02
	#<LONG	\$06 05
	#>DP	\$00 00
	#>ABS	\$02 00
	#>LONG	\$05 04
	#^DP	\$00 00
	#^ABS	\$00 00
	#^LONG	\$04 00
	/DP	\$00 00
	/ABS	\$02 00
	/LONG	\$05 04
Direct Page (Zero Page)	DP	\$01
	<ABS	\$03
	<LONG	\$06
Absolute	DP	\$01 00
	ABS	\$03 02
	LONG	\$06 05
Absolute Long	>DP	\$01 00 00
	>ABS	\$03 02 00
	LONG	\$06 05 04
Direct Page Indexed	DP,X	\$01
	<ABS,X	\$03
	<LONG,X	\$06
Absolute Indexed	DP,X	\$01 00
	ABS,X	\$03 02
	LONG,X	\$06 05
	DP,Y	\$01 00
	ABS,Y	\$03 02
	LONG,Y	\$06 05
Absolute Long Indexed	>DP,X	\$01 00 00
	>ABS,X	\$03 02 00
	LONG,X	\$06 05 04
Direct Page Indirect	(DP)	\$01
	(<ABS)	\$03
	(<LONG)	\$06
Direct Page Indirect Long	[DP]	\$01
	[<ABS]	\$03
	[<LONG]	\$06

Direct Page Indirect Indexed	(DP),Y	\$01
	(<ABS),Y	\$03
	(<LONG),Y	\$06
Direct Page Indirect	[DP],Y	\$01
	[<ABS],Y	\$03
	[<LONG],Y	\$06
Direct Page Indexed	(DP,X)	\$01
	(<ABS,X)	\$03
	(<LONG,X)	\$06
Stack Relative	DP,S	\$01
	<ABS,S	\$03
	<LONG,S	\$06
Stack Relative Indirect	(DP,S),Y	\$01
	(<ABS,S),Y	\$03
	(<LONG,S),Y	\$06
Accumulator	A	
Block Move	DP,DP	
	ABS,ABS	
	LONG,LONG	

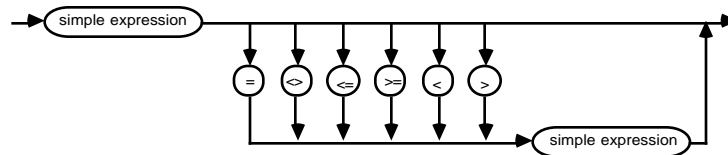
Table 16.1 Legal Operands for the 65816

Expressions

Whenever a number is allowed in an operand field, whether in a 65816 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



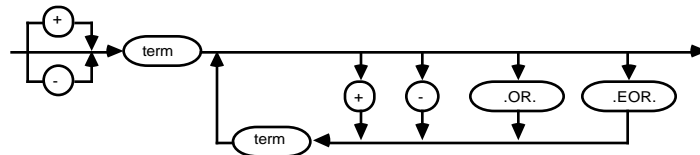
A simple expression is the customary arithmetic expression. Syntactically, this is expressed as an optional leading sign, a term, and, optionally, a +, -, .OR., or .EOR. followed by another term. Thus, logical comparisons have the lowest priority.

Some examples of valid expressions, using the above diagram:

Assembler Reference Manual

<u>expression</u>	<u>result</u>
2<4	1
2+1<>\$FFF	1
LOOPCOUNT=LOOPCOUNT+1	0

simple expression

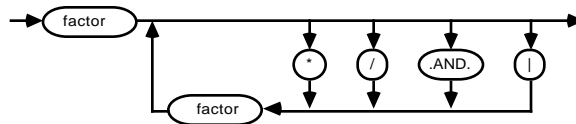


Examples using the above diagram:

<u>expression</u>	<u>result</u>
5+6	11
-3+2	-1
1.OR.0	1
3/4+6*2	12

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 $a \cdot (2^b)$. It is important to note that logical operators perform word comparisons, rather than bit-wise operations.

term

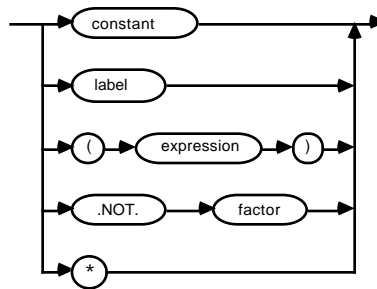


Examples:

<u>expression</u>	<u>result</u>
3	3
3/4	0
1+2*3	7

A factor is a constant, label, or 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. The * character indicates the current location counter. It resolves to the address of the first byte of the instruction.

factor



Examples:

expression

6*7=42

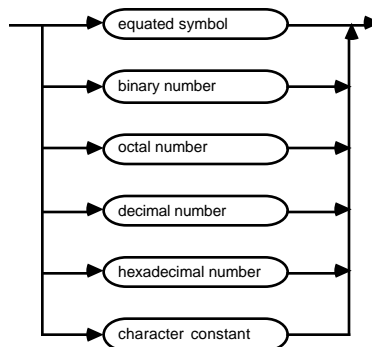
.NOT. (4+6=10)

result

1

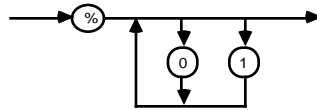
0

constant



Assembler Reference Manual

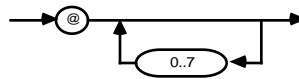
binary number



Examples:

<u>binary constant</u>	<u>decimal equivalent</u>
%0	0
%1	1
%10	2
%10100101	165

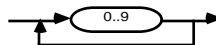
octal number



Examples:

<u>octal constant</u>	<u>decimal equivalent</u>
@6	6
@7	7
@10	8

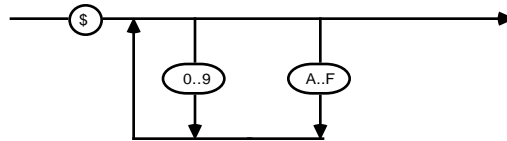
decimal number



Examples:

3
457

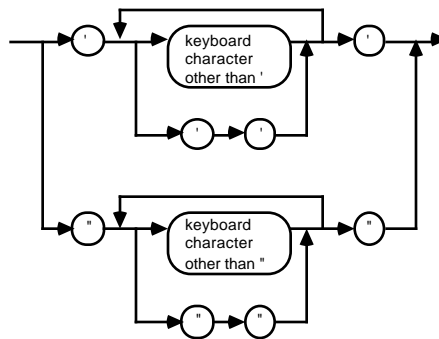
hexadecimal number



Examples:

<u>hexadecimal constant</u>	<u>decimal equivalent</u>
\$9	9
\$A	10
\$B	11
\$89ABCD	9022413

character constant



Examples:

<u>character constant</u>	<u>decimal equivalent</u>
'a'	97
"A"	65
'This is a character constant'	1416128883
' ' ' ' ' '	39

The Comment Field

In-line comments can start in any column past the first space after the operand field. If an instruction does not accept 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. Comments do not need to be preceded by a semicolon character.

Chapter 19

Assembler Directives

This chapter describes the assembler directives used in ORCA that are not designed for use in macros. Macro directives and conditional assembly directives are covered in the next chapter. Each directive has its proper syntax and use given. For easy reference, the directives are listed in alphabetic order.

Introduction To Assembler Directives

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. An underscored operand gives the directive's default setting. For example,

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

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

<u>name</u>	<u>description</u>
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 Chapter 18.

lab	A label. Labels start with an alphabetic character, underscore or tilde and are followed by zero or more alphanumeric characters, tildes and underscores. All of the characters are significant.
opcode	One of the operation codes recognized by the assembler, e.g. LDA, START.
pathname	A full or partial path name. For partial path names, the current prefix is used.
segname	The name of a load segment, i.e. a name that has appeared as the operand of a START, PRIVATE, PRIVDATA, 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.

ABSADDR

Show Absolute Addresses

[lab] ABSADDR ON|OFF [comment]

This directive gives the assembler's best guess at where the final code will rest. The guess shows up as a three-byte number to the left of the output normally printed by the assembler—all of the old output is moved seven columns to the right to make room. If no other indication of a final location is available, the assembler assumes \$00000. An ORG can change the value.

The values shown will have no bearing on reality unless the program is executed under ProDOS 8 or is ORGed. Keep in mind that ORGing a program severely handicaps the memory management scheme used on the Apple IIGS. Without a compelling reason to ORG your program (ROM code is the only one that comes to mind), using an ORG in an Apple IIGS program is a mistake. For that reason the values this directive lists cannot be relied upon in most programs.

Finally, keep in mind that the listing must be from a full assembly, and that the link must be a simple link, not one where the code produced is used as a library.

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, PRIVATE, DATA or PRIVDATA 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 two. 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.

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.

APPEND**Append a Source File**

```
[lab]    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.

CASE**Specify Case Sensitivity**

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

The CASE directive allows you to make the assembler case sensitive. If you specify CASE ON, subsequent label definitions and uses are case sensitive. See also OBJCASE.

CODECHK**Disable Code Bank Checks**

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

Normally, the assembler tries to check for errors where an absolute addressing mode is used with a label that appears in another bank or load segment. This directive lets you turn that error checking off. The instructions affected are JMP and JSL.

For example, if a JSR THERE is found, and THERE is in a different load segment from the JSR instruction, the assembler will flag an error – it expects you to use a JSL instead of a JSR. Coding CODECHK OFF before the JSR is encountered will suppress the error message.

See also DATACHK.

COPY**Copy a Source File**

```
[lab]    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 quite large.

DATA Define Data Segment

```
lab          DATA [segname]          [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 areas 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.

The *segname* in the operand field specifies the load segment. See the section on run-time segmentation for details.

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

DATACHK Disable Data Bank Checks

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

Normally, the assembler tries to check for errors where an absolute addressing mode is used with a label that appears in another load segment. This directive disables that check for all instructions except JSR and JMP - that is, for all instructions that use the data bank register (the B register) for long addressing, rather than the code bank register (the K register).

For example, if a LDA THERE is found, and THERE is in a different load segment from the LDA instruction, the assembler will flag an error - it expects you to override the addressing mode, as in LDA >THERE. Coding DATACHK OFF before the LDA instruction suppresses the error message.

See also CODECHK.

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 defined with 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 in the expression. The symbols have the same use as described for the immediate addressing mode. 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	-140737488355328	140737488355327
7	-36028797018963968	36028797018963907
8	-9223372036854775808	9223372036854775807

Here are some examples:

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'	AB 12
dc i5'3'	03 00 00 00 00
dc 2i3'1,2'	01 00 00 02 00 00
	01 00 00 02 00 00

Assembler Reference Manual

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 bytes. If *x* is omitted, the A DC defaults to two-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 ensure that any library subroutines that will be needed to execute that instruction are linked.

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

Code

```
dc      r'LIBRARY'
```

Soft Reference

```
S[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 executable program produced by the link editor will resolve the value 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:

Chapter 20: Macro Language and COnditional Assembler Directives

<u>Code</u>	<u>Value</u>
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:

<u>Code</u>	<u>Value</u>
dc	2b'01 01 01 10' 56 56
dc	b'11111111' 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:

<u>Code</u>	<u>Value</u>
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 IIGS text 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.

Assembler Reference Manual

This format is compatible with the IEEE floating-point standard, and is also used by the SANE tools. See the IEEE directive for a way to get five-byte floating-point numbers compatible with Applesoft.

Floating-point numbers are generally stored least-significant-byte first; this is the format used by most floating-point packages, including the ORCA/M 4.1 Floating-point Libraries (ProDOS 8 ORCA/M), as well as by a number of floating-point cards, such as the 68881. See NUMSEX for a way to change this order of the bytes.

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

<u>Code</u>	<u>Value</u>
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.

<u>Code</u>	<u>Value</u>
dc d'3,-3,.35E1,6.25 E-2'	4008000000000000 C008000000000000 400C000000000000 3FF0000000000000

Extended Precision Floating Point

nE'float-number[,float-number...]'

This is identical to F, except that a SANE extended 10-byte number is generated with a fifteen-bit exponent and a sixty-four-bit mantissa. Numbers can range from about -1E-4932 to 1E+4932. The mantissa is accurate to slightly more than 19 decimal digits. The exponent is stored most significant byte first.

<u>Code</u>	<u>Value</u>
dc e'3'	00 00 00 00 00 00 00 C0 00 40
dc e'-3'	00 00 00 00 00 00 00 C0 00 C0

Chapter 20: Macro Language and COnditional Assembler Directives

```
dc      e'.35I1'          00 00 00 00
                                00 00 00 E0
                                00 40
```

DIRECT

Set Direct Page Location

```
[lab]    DIRECT expression    [comment]
[lab]    DIRECT OFF          [comment]
```

The **DIRECT** directive is used to set the value of the Direct Page register, to be used internally by the assembler but not actually stored into the Direct Page register itself. The assembler must be able to evaluate the expression at assembly time. That is, the expression cannot contain any references to relocatable labels. Setting **DIRECT** to a value allows the assembler to automatically promote direct page values to absolute values when that is necessary. As an example, consider the following code:

```
SBC      L1,Y
```

where **L1** is a direct page location. Since there is no such addressing mode **DP,Y**, the assembler would normally flag this line as an error. If you had set **DIRECT** to a value, however, the assembler would generate code for the operand equal to the location of **L1** plus the value of **DIRECT**. The main use of **DIRECT** is for writing ProDOS 8 applications; in this case the value of **DIRECT** should be zero. This is because the zero page on the 8-bit Apple IIs starts at page zero in bank zero. On the Apple IIGS, the direct page is in bank zero, but not necessarily in page zero. The effect of **DIRECT** is cancelled by coding it with an operand of **OFF**. You may turn the directive on and off throughout your program.

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.

DYNCHK

Check References to Dynamic Segments

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

If **DYNCHK** is turned on, the only allowable opcode which refers to a label contained in a dynamic segment is **JSL**. If **DYNCHK** is turned off, all references to labels contained in dynamic segments are allowed. The linker and assembler work closely in building lists of references to resolve. At assembly time, the assembler tries to resolve all of the addresses it can: those derived from constant expressions, and those involving local labels, which become offsets from a base address, to be patched later by the linker. Unknown labels are left entirely for the linker to resolve. During link editing, the linker builds a jump table of global labels contained in dynamic segments. This jump table is passed to the system loader, to allow it to patch addresses at run-time. When **DYNCHK** is on, the linker will flag an error for any opcodes other than **JSL** which refer to labels

in its jump table (i.e. global labels found in dynamic segments). If DYNCHK is off, the linker will not generate these errors.

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 affect the code sent to the output file in any way.

END

End Program Segment

```
[ lab ]      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 some place 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. This allows a numeric value to be assigned to a name, with the name to be used instead of the number in further operands. This makes your code easier to understand.

You can also reference other values and the location counter with EQU. The expression you equate with the label does not have to be declared locally, nor does it have to be a fixed value. References to non-constant operands are passed on to the linker for resolution.

Some examples of using EQU are listed below:

```
two          equ  1+1
count        equ  globalCount+1
```

Another use of EQU is to assign a label to the current value of the location counter. The use:

```
label        equ  *
```

is identical to the directive ANOP. EQU can also be used to identify a variable within a group of data. For the example below, FOUR will point to the location of the floating-point value of 4. FOUR is equated with the fourth variable offset from the DC directive, and each floating-point value occupies four bytes.


```
four      equ    *+3*4
one       dc     f'1,2,3,4,5,6,7,8,9,10'
```

ERR**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**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.

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 that result in constants 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.

The most common use of GEQU is to declare fixed direct page and long addresses. Since these are constants, the assembler resolves them and is able to generate the proper operand lengths without explicit operand length specifiers. For example, the following code fragment shows how the assembler would choose operand lengths for a LDA instruction.

```
          addr    equ    4
          tools    equ    $E10000

A5 04          lda    addr
AF 00 00 E1    lda    tools
```

IEEE**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 affected; they always generate IEEE double-precision numbers.

INSTIME**Show Instruction Times**

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

Turning INSTIME on causes two columns to be inserted in the output listing immediately before the macro expansion column (where + characters mark lines generated by a macro). The first column will contain the number of CPU cycles required to execute an instruction. If the line is not an instruction, nothing appears in the column. If the number of cycles can vary with circumstances, such as page boundary crossings or size of registers, the next column contains an asterisk. Refer to a reference manual for details that can change the timing.

KEEP**Keep Object Module**

```
[lab]      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 file. This directive may only be used one time, and must appear before any code-generating statements.

KIND**Set Segment Kind**

```
[lab]      KIND expression         [comment]
```

The START, PRIVATE, DATA and PRIVDATA directives all cause the assembler to begin a new object segment. Object segments have a kind field which specifies certain characteristics about the segment. The four directives just mentioned allow you to set this kind field to the four most commonly used values. The kind field can, however, have up to 65536 distinct values, only some of which are defined at this time. This directive lets you set the kind field to any value from 0 to 65535. It is generally used right after the START, PRIVATE, DATA or PRIVDATA. It must be used inside of a segment.

Note that this directive bypasses the normal error checking of the assembler. You must ensure that the code in the segment is consistent with the segment kind you specify – the assembler cannot check for you.

For a list of the most common segment kinds, see Appendix B.

LIST**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**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 affected 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 sixteen-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
                                longi on
A9 01 00    lda    #1
A2 02 00    ldx    #2
                                longa off
                                longi off
A9 01      lda    #1
A2 02      ldx    #2
```

The status bit that the CPU uses at run-time must be set separately. To do so, use the REP and SEP instructions, or use the LONG and SHORT macros provided with ORCA.

LONGI**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.

MEM**Reserve Memory**

```
[lab]    MEM expression,expression  [comment]
```

The MEM directive is included for compatibility with older versions of ORCA/M. On the Apple IIGS, which has relocatable files, the MEM directive cannot be used. It will always result in a linker error.

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 or ASMLG command from the shell). The operand is evaluated to a one-byte positive integer.

The default value is zero.

MSB **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.

NUMSEX **Set the Byte Order of Floating-point Values**

[lab] NUMSEX ON|OFF [comment]

Most floating-point packages define numbers as they would appear when you write them: the most significant byte, followed by the next most significant byte, ..., down to the least significant byte. Apple's numeric package, SANE, defines floating-point numbers in the reverse, from the least significant byte up to the most significant byte. If NUMSEX is set to on, the assembler will generate floating-point constants from most significant byte down to the least significant byte.

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.

OBJCASE **Specify Case Sensitivity In Object Files**

[lab] OBJCASE ON|OFF [comment]

The OBJCASE directive allows the case sensitivity of the object module to be separately controllable from the case sensitivity of the language. Regardless of the setting of the CASE directive, symbols are internally maintained in the case they were defined with. When definitions

and references are written to the object module, the setting of the OBJCASE directive determines if they will be written in all uppercase (OBJCASE OFF, or case insensitive) or mixed uppercase and lowercase (OBJCASE ON).

Because the settings of CASE and OBJCASE should almost always be the same, changing CASE sets OBJCASE to the same value. If they are to be set differently, be sure and set OBJCASE *after* setting CASE.

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 relocatable. The ORG directive can be used to start the program at a fixed 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 executable file is built.

Note that using ORG in this fashion cripples the memory management scheme of the Apple II GS. If at all possible, avoid the use of the ORG directive!

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**Send Output to Printer**

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

If PRINTER ON is coded, output is sent to the printer. If the option is turned off, output is sent to the video display.

PRIVATE**Define A Private Code Segment**

```
lab      PRIVATE [segname]         [comment]
```

One feature that is very useful when doing separate compilation is the ability to hide a symbol from other source files. For C aficionados, this corresponds to static variables. The effect is to make a symbol global within the compilation, but to not make it available in code produced by other compiles or assemblies. PRIVATE and PRIVDATA allow this in the assembler.

The PRIVATE directive is used exactly like a START directive, and for the same purposes. The name of the segment is global within the assembly that it appears, but cannot be accessed from code produced by another assembly and later linked in. Symbols defined by GEQU and ENTRY directives within the segment are also private to the assembly.

The *segname* in the operand field specifies the load segment.

Note that marking a label as private has an additional side benefit. Since the symbol can only be accessed from a single assembly, the linker is able to handle more than one global label with the same name. The restriction is that all but one be marked as private, and that all appear from different assemblies.

PRIVDATA**Define A Private Data Segment**

```
lab      PRIVDATA [segname]        [comment]
```

PRIVDATA works just like a DATA directive, except that the segment and all labels declared within the segment are private. See PRIVATE for a discussion of what it means for a symbol to be private.

The *segname* in the operand field specifies the load segment.

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 65816 instruction. The second time, it is not found in the op code table, so the assembler tries to expand it as a macro.

Chapter 20: Macro Language and COnditional Assembler Directives

```
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 40, but can be changed to any number from 1 to 255 by specifying the number in the operand field of this directive.

START

Start Subroutine

```
lab      START [segname]            [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 *segname* in the operand field specifies the load segment.

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

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.

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 listing device. If an operand was coded, the string used will be printed at the top of each page, immediately after the page number.

USING

Using Data Area

```
[lab]    USING lab                [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

Enable 65C02 Code

```
[lab]    65C02 ON|OFF           [comment]
```

The 65C02 is used in the Apple //c and enhanced Apple //e, 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

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 20

Macro Language and Conditional Assembly Directives

This chapter tells 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 with 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 cause an error message to be printed.

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 operation code field. If an operation code that the assembler is trying to identify matches the name of the macro, the assembler replaces the macro call in the source file with the instructions found in the body of the macro itself. The macro name may be any sequence of keyboard characters except blanks or the & character. It may contain any number of characters.

Consider the following simple macro as an example. It is used to print a character on the screen. The Apple IIGS Text Tools has a tool call which can be used to write a character to the current output device. Since the tool call is a three-step process, it is inconvenient to try to remember all the steps, as well as the correct hex values every time a character needs to be written.

Assembler Reference Manual

(The steps include pushing the two-byte character onto the stack, loading the X register with the function number and tool number, and executing a long jump to the toolbox.) To remedy this, a macro may be defined, using a mnemonic name for the tool call:

```
macro
putchar
pha
ldx    #$180C
jsl    $E10000
mend
```

Putchar is created using the text editor, and is placed in a file named PUT.MACROS. It can now be used as a new instruction:

```
list    on
gen     on
mcopy   PUT.MACROS
keep    stuff
Main    start
phk
plb
lda     #'A'
putchar
lda     #0
rtl
end
```

Following the putchar instruction, the assembler includes the macro expansion, yielding the sequence

ORCA/M Asm65816 2.0

```
0001 0000          list on
0002 0000          gen  on
0003 0000          mcopy put.macros
0004 0000          keep stuff
0005 0000          Main start
0006 0000 4B       phk
0007 0001 AB       plb
0008 0002 A9 41 00 lda #'A'
0009 0005          putchar
          0005 48          + pha
          0006 A2 0C 18    + ldx  #$180C
          0009 22 00 00 E1 + jsr   $E10000
0010 000D A9 00 00   lda  #0
0011 0010 6B       rtl
0012 0011          end
```

```
12 source lines
1 macros expanded
3 lines generated
```

in the output listing. The new instruction `PUTCHAR` may be used as many times as desired anywhere in a source program provided the `MCOPY PUT.MACROS` directive is also included. The assembler always prints the name of the macro (in this case `PUTCHAR`) to show how instructions that follow were generated; the `PHA`, `LDX`, and `JSL` instructions are the only part of the macro expansion that actually generates code. The `+` character at the beginning of the lines containing these instructions 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, but can contain all other valid `ORCA` directives and 65816 instructions.

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 are implicitly defined by appearing on the macro definition line. Character type symbolic parameters 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    putchar    &c1
&lab    lda         &c1
        pha
        ldx         #$180C
        jsl         $E10000
        mend
```

Using the new macro, you do not have to load the accumulator with the character to print before using the macro. It is called from a source program as follows.

Assembler Reference Manual

```

        .
        .
        beq      L1
        putchar # 'A'
        bra      L2
L1      putchar # 'B'
L2      lda      #0
        .
        .

```

At assembly time, the code shown below 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:

0008	0002	F0	0D			beq	L1
0009	0004					putchar	#'A'
	0004	A9	41 00	+		lda	#'A'
	0007	48		+		pha	
	0008	A2	0C 18	+		ldx	#\$180C
	000B	22 00 00	E1	+		jsl	\$\$E10000
0010	000F	80	0B			bra	L2
0011	0011				L1	putchar	#'B'
	0011	A9	42 00	+L1		lda	#'B'
	0014	48		+		pha	
	0015	A2	0C 18	+		ldx	#\$180C
	0018	22 00 00	E1	+		jsl	\$\$E10000
0012	001C	A9	00 00		L2	lda	#0

The reason that &C1 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	putchar	&c1,&c2
&lab	lda	&c1
	pha	
	ldx	#\$180C
	jsl	\$E10000
	lda	&c2
	pha	
	ldx	#\$180C
	jsl	\$E10000
	mend	

Chapter 20: Macro Language and COnditional Assembler Directives

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:

```
0008 0002                                putchar #'A',#'B'
      0002 A9 41 00      +                lda    #'A'
      0005 48              +                pha
      0006 A2 0C 18      +                ldx    #$180C
      0009 22 00 00 E1 +                jsr    $E10000
      000D A9 42 00      +                lda    #'B'
      0010 48              +                pha
      0011 A2 0C 18      +                ldx    #$180C
      0014 22 00 00 E1 +                jsr    $E10000
```

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 (the beginning '&' is *not* typed), followed by an equal sign, and the value to assign it. For example, a call to the PUTCHAR macro could be coded as:

```
0008 0002                                putchar c2=#'F',c1=#'Z'
      0002 A9 5A 00      +                lda    #'Z'
      0005 48              +                pha
      0006 A2 0C 18      +                ldx    #$180C
      0009 22 00 00 E1 +                jsr    $E10000
      000D A9 46 00      +                lda    #'F'
      0010 48              +                pha
      0011 A2 0C 18      +                ldx    #$180C
      0014 22 00 00 E1 +                jsr    $E10000
```

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.

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

Assembler Reference Manual

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 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

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. 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.

The directives LCLA, LCLB, and LCLC are used to explicitly define local symbolic parameters. The directives GBLA, GBLB, and GBLC are used to explicitly define 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 the name of the symbolic parameter. Subscripts can range from 1 to 255. Only a single subscript is allowed. Subscripts can range from 1 to 255. A symbolic parameter used as a subscript for another symbolic parameter cannot be subscripted. Some examples are:

lcla	&a1	define local arithmetic sym. parm. &a1
gbla	&num(10)	define global arith. sym. parm. array &num
lclb	&log(3)	define local boolean sym. parm. array &log
gblb	&true	define global boolean sym. parm. &true
lclc	&str	define local character sym. parm. &str
gbld	&names(80)	define global char. sym. parm. array &names

Symbolic parameter definition statements are not printed in the output listing unless they contain errors or TRACE ON has been coded.

Values are explicitly assigned to symbolic parameters with the SETA, SETB, and SETC directives. The label field contains the name of the symbolic parameter, the opcode field contains a SETx directive whose type (A, B, or C) matches the type used to define the parameter, and the operand field contains the value to assign to the symbolic parameter. Assuming the definitions above, the following statements initialize the symbolic parameters:

Chapter 20: Macro Language and COnditional Assembler Directives

```
&a1      seta      2          set &A1 to 2
&num(3)  seta      10         set the 3rd value of the array &NUM to 10
&log(&a1) seta      0          set the 2nd value of the array &LOG to false
&str     setc      'hey, you!' set &STR to the string 'hey, you!'
```

Using the SETx directives, their values may be set and reset during macro expansions, resulting in an extremely powerful conditional assembly capability.

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.

<u>name</u>	<u>type</u>	<u>value(s)</u>
&art	A	\$FE
&bin(2)	B	1 (true) 0 (false)
&char	C	'LABEL'
&char2(3)	C	'STRING1' ' ' (null string) 'A'

Below left are instructions as typed in a macro file, 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.

Predefined Symbolic Parameters

The assembler contains five predefined symbolic parameters which you can use in your programs. 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. For example, the macro

Assembler Reference Manual

```
macro
demo.syscnt  &c1
lclc        &c2
&c2         setc  &c1.&syscnt
mend
```

would set &C2 to a different value each time it is called, regardless of the parameter &C1.

Three predefined character type symbolic parameters are available. &SYSNAME expands to the name of the current segment. It is useful in debug macros, which can automatically indicate what segment they are in. &SYSDATE expands to the date when the assembly started, and &SYSTIME to the time. The format for the date is DD MMM YY, while the format for the time is HH:MM.

For example, the comment

```
!      &systime &sysdate
```

would expand to

```
!      20:04 01 May 90
```

if the program in which it appeared was assembled at 8:04PM on May 1, 1990. These directives can be used to time and date stamp program executions, as in

```
dc      c'Prog X run at &SYSTIME on &SYSDATE'
```

The ASCII string will then appear in the executable program image.

The last predefined symbolic parameter is named &SYSOPR. It can only be used within a macro, and must appear in the macro definition line as the only parameter. It is of type character, and is set to the entire operand field of the macro call, beginning with the first non-space character following the call. The operand field of the macro call must contain a value or an error will result. An example of its use:

```
macro
demo.sysopr  &SYSOPR
; '&SYSOPR'
mend
```

if called in a program with these lines

```
demo.sysopr  Hey, you! How do you like programming the Apple IIGS?
demo.sysopr  oh...I guess I like it pretty well...
```

would result in the following expansions:

```
'Hey, you! How do you like programming the Apple IIGS?'
'oh...I guess I like it pretty well...'
```

&SYSOPR's main use is to allow the programmer a way to parse his own macro operands. It was used extensively, for example, in writing Byte Works' *Merlin-to-ORCA Source Code Translator*.

Sequence Symbols

The conditional assembly branch instructions AGO and AIF must have some place 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 an array of values has been implicitly 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. In the example below, the character symbolic array &ADR is assigned the values passed on the command line in the call to the ST1 macro. The arithmetic symbolic parameter &CNT is used internally to perform a loop to assign the values. The loop ends when &CNT is greater than the COUNT of &ADR, set to the number of subscripts as determined from the macro call:

Macro Definition:

```

macro
&lab      stl      &adr
          lcla      &cnt
&cnt      seta      1
&lab      lda      #1
.top
          sta      &adr(&cnt)
&cnt      seta      &cnt+1
          aif      &cnt<=C:&adr,^top
          mend

```

Assembler Reference Manual

Macro Use:

```
          stl      (N1,N2,N3)
+         lda      #1
+         sta      N1
+         sta      N2
+         sta      N3
```

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. The macro below is used to define a "Pascal string;" that is, a string whose first byte is its length, followed by the characters forming the string:

Macro Definition:

```
          macro
&lab      dw      &str
&lab      dc      il'1:~&syscnt
~&syscnt  dc      c"&str"
          mend
```

Macro Use:

```
msg       dw      'Hello, world.'
+MSG      dc      il'1:~2'
+~2       dc      c"Hello, world."
```

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.

ABSADDR	CASE	CODECHK	DATACHK	DYNCHK
ERR	EXPAND	GEN	IEEE	INSTIME
LIST	LONGA	LONGI	MSB	NUMSEX
OBJCASE	PRINTER	SYMBOL	TRACE	65C02
65816				

Chapter 20: Macro Language and COnditional Assembler Directives

Macro Definition:

```
macro
&lab    stz1    &adr
&lab    aif     S:longa=0,.A    Check if using long accumulator
        sep     #$20           Set to short if using long
.A
        stz     &adr           Store 1-byte zero to parameter
        aif     S:longa=0,.B    Reset to long if had set to short
        rep     #$20
.B
        mend
```

Macro Use: (long accumulator)

```
        stz1    cnt
+       sep     #$20           Set to short if using long
+       stz     cnt           Store 1-byte zero to parameter
+       rep     #$20
```

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:

Type	Meaning
A	Address Type DC Statement
B	Binary Type DC Statement
C	Character Type DC Statement
D	Double-Precision Floating-Point Type DC Statement
E	Extended 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.

Assembler Reference Manual

The macro below detects if its passed parameter is a 4-byte floating-point number:

```
macro
&lab    phf      &adr
        lclc     &C
&C      setc     T:
        aif      &C&adr="F",.A
        mnote    "Must be floating point",16
.A
&lab    lda      &adr+2
        pha
        lda      &adr
        pha
        mend
```

Macro Use:

```
        phf      num
+       lda      num+2
+       pha
+       lda      num
+       pha
        .
        .
        .
num     dc        f'4.3'
```

Conditional Assembly and Macro Directives

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 unless the label is coded.

Examples of macro definitions and how they use conditional assembly directives can be found in the macro library.

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.

Example:

```

        lcla    &n1
&n      seta    3000
.top
        actr    100
&n1     seta    &n1-1
        aif     &n1, ^top

```

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. If the sequence symbol is not found, 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
! This line is 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-param 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-param AMID string,express,express    [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:

<u>instruction</u>	<u>resulting string</u>
&char amid 'TARGET',2,3	ARG
&char amid 'TARGET',5,3	ET
&char amid 'TARGET',7,3	null string

ASEARCH

Assembler String Search

```
sym-param ASEARCH string,string,express [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.

Assembler Reference Manual

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:

<u>instruction</u>	<u>resulting value</u>
&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 an array by following the name with the number of subscripts enclosed in parentheses. If used, the subscript must be in the range 1 to 255.

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 in which it is defined, including inside macros and in the source file itself.

The symbolic parameter can be declared as an array by following the name with the number of subscripts enclosed in parentheses. If used, the subscript must be in the range 1 to 255.

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 in which it is defined, including inside macros and in the source file itself.

The symbolic parameter can be declared as an array by following the name with the number of subscripts enclosed in parentheses. If used, the subscript must be in the range 1 to 255.

GEN 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.

LCLA Declare Local Arithmetic Symbolic Parameter

```
LCLA sym-param [comment]
```

Defines the arithmetic symbolic parameter *sym-param*. The symbolic parameter is valid only in the segment or macro in which it is defined.

The symbolic parameter can be declared as an array by following the name with the number of subscripts enclosed in parentheses. If used, the subscript must be in the range 1 to 255.

LCLB Declare Local Boolean Symbolic Parameter

```
LCLB sym-param [comment]
```

Defines the boolean symbolic parameter *sym-param*. The symbolic parameter is valid only in the macro or segment in which it is defined.

The symbolic parameter can be declared as an array by following the name with the number of subscripts enclosed in parentheses. If used, the subscript must be in the range 1 to 255.

LCLC

```
LCLC sym-parm [comment]
```

Defines the character symbolic parameter *sym-param*. The symbolic parameter is valid only in the segment or macro in which it is defined.

The symbolic parameter can be declared as an array by following the name with the number of subscripts enclosed in parentheses. If used, the subscript must be in the range 1 to 255.

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.

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, the 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.

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.

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 to 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.

Example:

```

      macro
&lab  check      &n1,&n2,&n3
      aif      &n1,.A
      mexit
.B      aif      &n2,.B
      mexit
.B      clc
      lda      &n1
      adc      &n2
      sta      &n3
      mend

```

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.

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 Symbolic Parameter**

```
sym-param SETA expression    [comment]
```

The operand field is resolved as a four-byte signed integer and assigned to the symbolic parameter in the label field.

Examples:

```
&num    seta    4
&n(&num) seta    &num2+label*4
```

SETB**Set Boolean Symbolic Parameter**

```
sym-param 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.

Example:

```
&flag      setb      a<&num
```

SETC

Set Character Symbolic Parameter

```
sym-param 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:

```
&strng(4) setc      &name1
&str      setc      '&fkename'+'.obj'
```

TRACE

Trace Macros

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

Most conditional assembly directives 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.

Chapter 21

Introduction to the Macro Libraries

The ORCA 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. The types of macros found in the libraries are listed below.

- Input/Output macros.
- Integer math macros.
- Miscellaneous macros.
- Operating system macros.
- Shell calls.
- Tool set macros.

Since the system is fairly automatic, the macros can be learned as if they were simply extensions to the 65816 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 link the program in the normal way.

The ORCA macro files are located in two directories, AInclude and ORCAInclude. AInclude contains all of the macros and equate files written by Apple Computer for use with APW and ORCA/M. The ORCAInclude folder contains an advanced set of tool and GS/OS interface macros that support passing parameters to the tools on the macro call line, as well as a number of utility macros. These macros are the ones described in the chapters that follow.

If you have installed ORCA/M on a hard disk, the AInclude and ORCAInclude folders will be installed in the libraries folder of your ORCA system; you can get at these macros using the prefixes 13:AInclude or 13:ORCAInclude. From floppy disks, these folders are on your extras

disk. On the floppies, the path names for these folders are :ORCA.Extras:Libraries:AIinclude and :ORCA.Extras:Libraries:ORCAInclude.

Macro files are standard source files. You can edit the files to check names and parameters, but you should probably not make changes to the macros. Instead, make any custom changes to a copy of the macro that you use with a specific program.

The documentation of the macros is divided into five chapters. Calling the GS/OS operating system, as well as the built-in tool box, is discussed later in this chapter. This chapter also covers 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 addressing modes which are common throughout the macro libraries. The next four chapters discuss the macros themselves. Macros are presented by topic. The first section deals with the mathematics macros, the second with input and output, the third with shell macros, and the last chapter with those macros that did not fit into one of the previous chapters.

GS/OS Macros

GS/OS provides a very regular set of subroutine calls which allow most of the common functions of disk interface to be performed. The macros in this section are primarily designed to "hide the ugly," freeing you from looking up the op codes associated with the operating system calls and helping avoid possibly disastrous results of misplacing the addresses of the control blocks usually associated with these calls.

Macro Naming Conventions

The operating system calls are in the macro file ORCAInclude:M16.GS.OS. It contains calls to both ProDOS 16 and GS/OS. The calls to ProDOS 16 use the same name as you would find in the *ProDOS 16™ Reference Manual*, with an underscore character before the name.

Unlike the tool macros, Apple has changed the style of the ProDOS macros over the years. While the current versions of the ProDOS macros all start with an underscore character, there was a time when the underscore character was not used. To maintain compatibility with programs written with the older style of ProDOS macros the M16.GS.OS file includes a full set of macros with the underscore character, and another full set that does not include the underscore.

The GS/OS operating system has most of the same calls that were in ProDOS, so there needs to be some way of distinguishing between the GS/OS macros and the ProDOS macros. Unfortunately, the mechanism used changed fairly early on, so we again have two distinct macro naming conventions. One convention placed the characters OS before each of the macro names, while another convention placed GS after the names of the calls. The current trend seems to be to use the GS suffix, but the M16.GS.OS file actually supports both conventions. Once again, underscores have been added; the macros that use the GS suffix all come both with and without the underscore character.

There is one other area of change in the macros. Early versions of the ProDOS and GS/OS macros used underscores in complex call names, like GET_PATH_NAME. These underscores do not appear in the call names listed in the *ProDOS 16™ Reference Manual*, or the *GS/OS™ Reference Manual*. When these names have been used in older macro files, either those supplied by Apple Computer or those supplied by the Byte Works, the old names are still in the M16.GS.OS macro file, but there are newer versions of each of these macros that do not include the extra underscore characters.

Putting all of this together, you should be able to use M16.GS.OS to generate macros for any existing program, whether that program used older versions of the ProDOS or GS/OS macros or

Chapter 21: Introduction to the Macro Libraries

the latest version. For new programs, type the call names exactly as they are in the reference manuals, with no extra imbedded underscore characters. Precede each macro name with an underscore character, and add the characters GS after any GS/OS call.

Inside the GS/OS Macros

Each call to the operating system consists of a JSL to \$E100A8, followed by a two-byte command number and a four-byte address of the control record. The control record is a section of code which defines data fields. All communication is through the control record; this is where the operating system gets the inputs for the call, as well as where the outputs are placed. Each of the operating system macros requires a single parameter, the absolute address of the control record. Calls to either operating system are distinguished in the command number. Calls to ProDOS 16 use a command number of the form \$00xx, while those to GS/OS use a command number of the form \$20xx.

As an example, a CREATE call to GS/OS is demonstrated below.

```
_CreateGS crRec      call GS/OS to create new file
      bcs      error  branch to err handler if error returned
```

The code below defines the record used with the create call. Notice the data fields declared. Almost every operating system call requires data fields, defined in a specific order, as part of the control record.

```
crRec      anop      Create definition
crPCount   dc        i'7'
crName     dc        a4'pathName'  pointer to path name of new file
crAcc      dc        i'$00C3'      define access flags
crType     dc        i'4'          ASCII text file
crAux      dc        i4'0'         sequential access
crStore    dc        i'1'          standard file
crEOF      dc        i4'$10000'    init. file size to 64K bytes
crReSrc    dc        i4'0'         not an extended file
crEnd      anop

pathName   dOSIn     'myFile'      name of file to create
```

Operating system calls must be made in native mode. Registers can be either eight or sixteen bits. If you were using short mode before the call, you will be in short mode following the call.

Tool Set Macros

The Apple IIGS has an extensive set of tool calls. Many of the ORCA macros actually translate into tool calls. The macro file M16.TOOLS contain macros to make the tool calls described in volumes one through three of the *Apple IIGS Toolbox Reference Manual*. The calls are organized alphabetically by tool set name. The names given to the tool call macros are derived from the *Apple IIGS Toolbox Reference Manual*. Each name is preceded with a tilde (~). For example, the Control Manager call CtlBootInit has a macro named ~CtlBootInit.

These tool macros allow you to pass parameters to the tools pretty much the way you would in a high-level language, putting the parameters on the same line as the macro call. For example, to draw a square you could use

Assembler Reference Manual

```
~MoveTo #10,#10
~LineTo #10,#100
~LineTo #100,#100
~LineTo #100,#10
~LineTo #10,#10
```

The tool macros support several addressing modes, including immediate addressing (shown in the example, above), absolute addressing, direct page addressing, long absolute addressing, indirect addressing, and long indirect addressing. There is also a special operator, *, to tell the macro that a parameter is already on the stack, and does not need to be loaded. For example, this sequence is equivalent to the first ~MoveTo call from the last example:

```
lda      #10
pha
pha
~MoveTo *,*
```

For a detailed explanation of the various addressing modes you can use, see the next section.

Some tools are functions, returning a result on the stack. For these calls, you must push space on the stack before making the tool call, and the result is left on the stack when the macro is complete. Here's a sample showing a call to GetPort showing how this works:

```
pha                make room on the stack
pha
~GetPort           get the grafPort
pl4    port        pull the result and save in port
```

All of the names used in the tool macro file match the names in the toolbox reference manuals exactly; you just add the tilde character in front of the name. Using the macro does not effect the register contents as returned by the tool in any way; the accumulator still has the error code, and the carry flag still indicates if an error was flagged.

There is another complete set of tool macros in the AInclude folder. These are an older style of tool macro that does not support parameters on the macro call line. With these macros, you must push all parameters on the stack before making the macro call. These macros are created and maintained by Apple Computer Inc., and are included in ORCA/M for your convenience.

The Apple tool macros are contained in a series of files, one for each tool. The macro names match the toolbox reference manuals, but again, you need to put another character before the name; this time, the character is the underscore character (_).

In general, tool calls using the Apple tool macros are made by:

1. Pushing to make room on the stack for any output values returned by the call.
2. Pushing any needed parameters, in the order specified in the toolkit reference manual.
3. Making the call by simply coding the correct macro name in the opcode field.
4. Pulling any values returned by the tool call from the stack.

Here's the example showing how to draw a square, reworked to use the Apple tool macros:

Chapter 21: Introduction to the Macro Libraries

```
ph2      #10
ph2      #10
_MoveTo
ph2      #10
ph2      #100
_LineTo
ph2      #100
ph2      #100
_LineTo
ph2      #100
ph2      #10
_LineTo
ph2      #10
ph2      #10
_LineTo
```

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 three addressing modes supported by the macros: immediate, absolute, and indirect.

Immediate addressing is available for 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      #5000000000
```

would write the approximate population of the Earth to standard output.

Absolute addresses are coded as a number, label, or expression, using the same rules as absolute addresses in instruction operands. An absolute address designates the memory location to use as a source or destination by the macro.

```
put8      big
```

Two-byte 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.

To perform long indirect addressing, use this format:

```
MUL4      [P1],[P2]
```

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

Assembler Reference Manual

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 two-byte integers, while PUTS writes strings.

<u>Character</u>	<u>Type</u>
2	signed two-byte integer
4	signed four-byte integer
8	signed eight-byte integer
C	character
S	ORCA-type string; 2 length bytes followed by sequence of characters
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, in the documentation, for anything else. Of course, in your programs, you can use either uppercase or lowercase letters, and in our sample code we tend to use lowercase letters exclusively.

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 complement 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 represented by eight-byte integers is from -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.

Chapter 21: Introduction to the Macro Libraries

Control characters have different 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. Programs running under ORCA use the GS/OS .CONSOLE driver, so a wide variety of control characters are supported.

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 255. The next byte contains the number of characters currently in the string; this ranges from 0 to the value of the maximum length. 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, because of the length fields.

Boolean Variables

Boolean variables require one byte of storage. They are either TRUE (non-zero) or FALSE (zero).

Memory Usage

All macros and subroutine libraries allocate work space from the stack. The program should allow at least 128 bytes of free stack space for use by the libraries. No other memory is used by the libraries.

Side Effects

Some of the macros have side effects. These effects are primarily manifested as destroying the contents of the registers; side effects are noted in the description of the macros.

Chapter 22

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 file M16.ORCA, located in the ORCAInclude directory in your libraries folder. You can access this file using the path name 13:ORCAInclude:m16.ORCA from your hard disk, or if you are using floppy disks, from :ORCA.Extras:Libraries:ORCAInclude:m16.ORCA.

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:

	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 uses indirect 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. If an overflow occurs, the V flag is set, otherwise it is cleared.

Coding Examples:

!	ADD4	{P1}, NUM1	adds the 4-byte integer pointed
!			to by P1 to the 4-byte integer
!			at NUM2, saving the result in
	ADD8	NUM1, #4	the location pointed to by P1
!			adds 4 to the 8-byte integer at
	ADD2	NUM1, NUM2, NUM3	NUM1
!			adds the 2-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
LAB    CMPL    NUM1 , NUM2
LAB    CMPW    NUM1 , NUM2

```

Operands:

LAB - Label.
 NUM1 - The first argument.
 NUM2 - The second argument.

Description:

The first integer is compared to the second 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 the setting of the status flags) is produced. The contents of all registers are lost.

Unlike the 65816 compare instruction, the comparisons performed by CMP2, CMP4 and CMP8 are 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.

The CMPW macro performs an unsigned compare of two 2-byte integers.

The CMPL macro performs an unsigned compare of two 4-byte integers.

Coding Examples:

```

                                CMP2    {P1} , #4           compare the 2-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 overflow flag is set.

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:

result	DIV4	#7,NUM1,NUM2	divides 7 by NUM1, placing the
!			in NUM2; the 3rd operand is
!			required, since the default
operand			
!			uses immediate addressing, which
!			cannot be used by a result
	DIV2	#7,#2,NUM1	an inefficient way to set 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 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 overflow flag is set.

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 flag is set.

Coding Examples:

pointed	MUL4	{P1} , #4	multiplies the 4-byte integer
!			to by P1 by 4
!	MUL2	NUM1 , NUM2 , NUM3	multiplies the 2-byte integers at
at			NUM1 and NUM2, saving the result
!			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.

Coding Examples:

```
!                RAN8    NUMBER                places an 8-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 Example:

```
        SIGN8  NUM1                replaces NUM1 with the result
```

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 overflow flag is set.

Coding Examples:

	SQRT2	#450, NUM2	places 21 in NUM2
	SQRT4	{P1}	replaces the 4-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 addressing, the contents of the Y register are also lost. The overflow flag is set if there is an overflow and cleared otherwise.

Coding Examples:

```
                SUB4    {P1} , NUM2 , {P2}      subtracts the 4-byte integer at NUM2
!                                                       from the 4-byte integer pointed at
!                                                       by P1, placing the result at the
!                                                       location pointed to by P2
```

Chapter 23

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 Apple IIGS Text Tools using the `_WriteChar` and `_ErrWriteChar` calls, and all input is received through the `_ReadChar` call. While running under the shell, the text tools are redirected to the GS/OS Console driver. For programs that run independently of the shell, be sure and start some form of text tool driver; the built-in Pascal driver is a good choice.

The macros in this section are contained in the file `M16.ORCA`, located in the `ORCAInclude` directory in your libraries folder. You can access this file using the path name `13:ORCAInclude:m16.ORCA` from your hard disk, or if you are using floppy disks, from `:ORCA.Extras:Libraries:ORCAInclude:m16.ORCA`.

ALTCH

Select Alternate Character Set

Forms:

```
LAB    ALTCH [ERROUT]
```

Operands:

LAB - Label.

ERROUT - Error out flag.

Description:

The ASCII control character `$0E` is sent to the output device. If the device is a CRT, this enables the alternate character set. Printers may also use this code; see your User's Manual for details. The contents of all registers are lost.

Normally, output is sent to standard out. If `ERROUT=T` is coded, output is sent to error out.

See the `NORMCH` macro for a way to reverse the effect.

Coding Examples:

```
ALTCH
ALTCH  ERROUT=T
```

BELL

Beep the Bell

Forms:

LAB BELL [ERROUT]

Operands:

LAB - Label.

ERROUT - Error out flag.

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 contents of all registers are lost.

Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

Coding Examples:

```
        BELL
        BELL    ERROUT=T
```

CLEOL

Clear to End of Line

Forms:

LAB CLEOL [ERROUT]

Operands:

LAB - Label.

ERROUT - Error out flag.

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. The contents of all registers are lost.

Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

Coding Examples:

```
        CLEOL
        CLEOL    ERROUT=T
```


CLEOS

Clear to End of Screen

Forms:

LAB CLEOS [ERROUT]

Operands:

LAB - Label.

ERROUT - Error out flag.

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. The contents of all registers are lost.

Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

Coding Examples:

```
CLEOS
CLEOS  ERROUT=T
```

COUT

Character Output

Forms:

LAB COUT CHAR[,ERROUT]

Operands:

LAB - Label.

CHAR - Character to write.

ERROUT - Error out flag.

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 contents of all registers are lost.

Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

Coding Examples:

COUT	#'.'	write a .
COUT	CH	write the character at CH
COUT	#' ',ERROUT=T	send a space to error out

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. They will receive all information through the `_ReadChar` call in the Apple IIGS Text Tools, which means that the input stream can be redirected to read from disk drives or other input devices.

N1 is used to compute the effective address where the variable read will be stored. It can be specified as an absolute address or an indirect 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 an ORCA string. The macros, and the types they input, are:

```

GET2    two-byte integer
GET4    four-byte integer
GET8    eight-byte integer
GETC    character
GETS    ORCA string; 2 length bytes followed by sequence of chars.

```

You can use the DSTR macro to initialize the area for a string read by setting its maximum expected length. GETS will store a carriage return (ASCII character \$0D) in the byte following the string's character bytes.

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 GETx macro call 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

Assembler Reference Manual

```
GET2  NUM1,CR=T
GET2  NUM2,CR=T
```

would read the numbers from two consecutive lines.
The contents of all registers are lost.

Coding Examples:

```
                GET2  INT
!
                GETC  CHAR,CR=T
!
RETURN
!
```

```
reads a 2-byte integer from the
keyboard
reads a character from the keyboard,
then skips characters until a
is typed
```

GOTOXY

Position Cursor On Screen

Forms:

LAB GOTOXY X,Y[,ERROUT]

Operands:

LAB - Label.

X - Column number, counting from 0.

Y - Row number, counting from 0.

ERROUT - Error out flag.

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 79 then the cursor is placed at the far right of the screen. Operands can be immediate or absolute. All register contents are lost.

Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

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 [ERROUT]

Operands:

LAB - Label.

ERROUT - Error out flag.

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. The contents of all registers are lost. Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

Coding Examples:

```
        HOME
HOME    ERROUT=T
```

NORMCH**Select Normal Character Set**

Forms:

LAB NORMCH [ERROUT]

Operands:

LAB - Label.

ERROUT - Error out flag.

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 contents of all registers are lost. Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

Coding Examples:

```
        NORMCH
NORMCH  ERROUT=T
```

PRBL

Print Blanks

Forms:

LAB PRBL NUM[,ERROUT]

Operands:

LAB - Label.

NUM - Number of blanks to print.

ERROUT - Error out flag.

Description:

NUM is the number of blanks to print. It must be a valid operand for a load instruction. If zero, 65536 blanks are printed. The contents of all registers are lost.

Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

Coding Examples:

```
PRBL #10          print 10 blanks
```

PUTx**Variable Output****Forms:**

```

LAB    PUT2   N1[,F1][,CR][,ERROUT]
LAB    PUT4   N1[,F1][,CR][,ERROUT]
LAB    PUT8   N1[,F1][,CR][,ERROUT]
LAB    PUTB   N1[,F1][,CR][,ERROUT]
LAB    PUTC   N1[,F1][,CR][,ERROUT]
LAB    PUTS   N1[,F1][,CR][,ERROUT]

```

Operands:

LAB - Label.

N1 - Location to place the variable read.

F1 - Field with.

CR - Carriage return flag.

ERROUT - Error out flag.

Description:

The PUTx macros are the standard way of writing information to external devices. They write all information to standard output or standard error output, depending on the ERROUT flag. All registers are lost in the process.

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 and indirect. 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".

As with the GETS macro, the string printed by PUTS is in ORCA format: the first byte holds the string's maximum possible length, the second byte contains its current length, and the subsequent bytes are the string's characters. You can use the DSTR macro to define and initialize the string.

Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

Chapter 23: Input and Output Macros

Coding Examples:

	PUT2	INT,CR=T	writes the 2-byte integer to the CRT followed by a carriage return
!			
!	PUTS	#'They''''re here...',#20	prints "They're here..." to the CRT, right-justified
!			
in			
!			a 20-byte field

PUTCR

Carriage Return

Forms:

```
LAB    PUTCR [ERROUT]
```

Operands:

LAB - Label.

ERROUT - Error out flag.

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 contents of all registers are lost.

Normally, output is sent to standard out. If ERROUT=T is coded, output is sent to error out.

Coding Examples:

```
PUTCR  
PUTCR  ERROUT=T
```

Chapter 24

Shell Calls

The shell acts as an interface and extension to GS/OS. The shell provides several functions not provided by the operating system; these functions are called exactly like GS/OS functions. Every time a program running under the ORCA shell issues a system call, the shell intercepts the call; if it is a shell call, the shell interprets it and acts on it. If it is a GS/OS call, the shell passes it on to GS/OS. This chapter describes all of the shell's system calls, here referred to as shell calls. The macros are in the file M16.SHELL.

Just as GS/OS extended ProDOS by allowing longer path names and creating parameter blocks with parameter counts while still using basically the same calls, the shell also has two levels of calls. The older shell calls follow ProDOS conventions, while the newer shell calls have parameter blocks that follow GS/OS conventions. The newer calls are formed from the old call numbers by oring the original call number with \$0040.

In the discussion that follows, the new, GS/OS style calls are documented. The older ProDOS style calls still appear in the macro file, but should not be used in new programs.

As with the GS/OS macros, to form the macro name from the call name shown in this chapter, and GS to the end. The correct macro name is shown in the model statement.

Making a Shell Call

Although shell calls are made exactly like GS/OS calls, this section does not provide all of the information relevant to GS/OS calls. GS/OS calls are described in the *GS/OS™ Reference Manual*.

The Control Record

A control record is a specifically formatted table that occupies a set of contiguous bytes in memory. It consists of a number of fields that hold information that the calling program supplies to the shell, as well as information returned by the shell to the caller.

Every shell call requires a valid control record, referenced as an absolute address in the operand of the macro call. You are responsible for constructing the parameter block for each call you make; the block may be anywhere in memory. Formats for individual parameter blocks accompany the detailed system call descriptions in this chapter.

Types of Parameters

Each field in a parameter block contains a single parameter. There are three types of parameters used by the shell: values, results, and pointers. Each is either an input to the shell from the caller, or an output from the shell to the caller.

A value is a numeric quantity, 1 or more bytes long, that the caller passes to the shell through the parameter block. It is an input parameter.

A result is a numeric quantity, 1 or more bytes long, that the shell places into the parameter block for the caller to use. It is an output parameter.

A pointer is the 4-byte address of a location containing data, code, an address, or buffer space in which the shell can receive or place data. The pointer itself is an input; that is, you always provide the pointer and reserve space for the data. The data pointed to may be either input by your program, returned by the shell, or both.

A given parameter may be both a value and a result.

Unless noted otherwise, each string in a control record is a GS/OS input or output string, as appropriate. A GS/OS input string is used for parameters that pass a string value to the shell call. A GS/OS input string consists of a length word followed by ASCII characters. A GS/OS output string is a buffer used to pass a string back from the shell; you should set up the pointer to the buffer in the control record before making the shell call. The output string starts with a two-byte buffer length. This buffer length is the total number of bytes occupied by the GS/OS output string, including the bytes used for the buffer length and length of the string. This value must be filled in before making the shell call. The shell will fill in the bytes after the buffer length with a string in the same format as a GS/OS input string, namely a length word followed by ASCII characters. The shell will return an error if the buffer is not large enough to hold a particular string value.

Register Values

There are no register requirements on entry to a shell call. The shell saves and restores all registers except the accumulator (A) and the processor status register (P); those two registers store information on the success or failure of the call. On exit, the registers have these values:

A	zero if call successful; if nonzero, number is the error code
X	unchanged
Y	unchanged
S	unchanged
D	unchanged
P	{see below}
DB	unchanged
PB	unchanged
PC	address of location following the parameter block pointer; if you are using the macros provided in ORCA to make the call, PC will point to the instruction following the shell call in your program

Unchanged means that the shell initially saves, and then restores when finished, the value the register had just before the shell call.

Chapter 24: Shell Calls

On exit, the processor status register (P) bits are

n	undefined
v	undefined
m	unchanged
x	unchanged
d	zero
i	unchanged
z	undefined
c	zero if call successful, 1 if not
e	zero

ChangeVector (\$014C)

Change a Shell Vector

Forms:

LAB ChangeVectorGS REC

Description:

Several subroutines called by the shell to perform standard operations may be changed to point to user supplied subroutines. This is usually done to support desktop programming environments that want to support the shell. The call replaces the current vector with a pointer to the user supplied subroutine, and returns the old value for subsequent restoration. The vectors that can be changed are:

vector use

- 0 Edit vector. A JSL is made to this location when a file is to be edited. Do a GetLInfo call to get the file name, entry point, and any error message passed by a compiler.
- 1 Console output vector. This subroutine is called by the ConsoleOut shell call to write a character directly to the console driver, bypassing I/O redirection and the Text Tools. The character to write is on the stack, pushed as a word. Your subroutine must be capable of being called with either 8- or 16-bit registers.
- 2 Stop vector. After cleaning up, the shell jumps to this location to abort a failed EXEC file. Normally, this is the shell's line editor, but a desktop program needs to change this vector so that control returns to the desktop program.

Parameter List:

0	pCount	
1		
2	reserved	
3		
4	vector	
5		
6	procPtr	
7		
8		
9		
A	oldProcPtr	
B		
C		
D		

Chapter 24: Shell Calls

<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type (range of values)</u>
\$00-\$01	pCount	parameter count	2-byte value [4] Parameter count; must be 4.
\$02-\$03	reserved	none	2-byte value [\$0000] Reserved for future use. Set this field to 0.
\$04-\$05	vector	number of vector to be changed	2-byte value [\$0000-\$0002] This is the number of the vector you wish to change. See the table above.
\$06-\$09	procPtr	pointer to user's subroutine	4-byte pointer [\$0000 0000– \$00FF FFFF] This is a pointer to the subroutine the shell will call.
\$0A-\$0D	oldProcPtr	pointer to shell's subroutine	4-byte pointer [\$0000 0000– \$00FF FFFF] The shell returns the old value for the vector in this location. You should restore the vector to this value before your program finishes.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```
ChangeVectorGS chRec
.
.
.
chRec      dc      i'4'
           dc      i'0'
chVector   dc      i'0'
chprocPtr  dc      a4'MyEditor'
chOldProcPtr ds    4
```

ConsoleOut (\$015A)**Write to the Console****Forms:**

```
LAB      ConsoleOutGS REC
```

Description:

The character is written to the console driver, bypassing I/O redirection. The principle reason for doing this is to send control characters to the console driver, such as cursor positioning characters or to turn the cursor on or off. By bypassing I/O redirection, the special characters are not written to files when the output is redirected.

Parameter List:

0	pCount
1	
2	char
3	

<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type (range of values)</u>
\$00-\$01	pCount	parameter count	2-byte value [1]
		Parameter count; must be 1.	
\$02-\$03	char	character to send to console driver	2-byte value [\$0000-\$00FF]

The character to send to the console driver. Only the least significant byte of the character field is actually used.

Possible Errors:

```
$04  Invalid parameter count
$53  Parameter out of range
```

Coding Example:

```

                ConsoleOutGS coRec
                .
                .
                .
coRec          dc      i'1'
coChar         dc      i'$13'
```


Direction (\$014F)**Check I/O Redirection****Forms:**

```
LAB          DirectionGS REC
```

Description:

A program can use this function to find out whether command-line I/O redirection has occurred. This function can be used by a program to determine whether to send form feeds to standard output, for example.

Parameter List:

0	pCount
1	
2	device
3	
4	direct
5	

Offset	Label	Parameter Name	Size and Type (range of values)
\$00-\$01	pCount	parameter count	2-byte value [2]

Parameter count; must be 2.

\$02-\$03	device	Device number	2-byte value [\$0000-\$0002]
-----------	--------	---------------	------------------------------

This parameter indicates which type of input or output you are inquiring about, as follows:

\$0000	Standard input
\$0001	Standard output
\$0002	Error output

\$04-\$05	direct	Direction	2-byte result [\$0000-\$0002]
-----------	--------	-----------	-------------------------------

This parameter indicates the type of redirection that has occurred, as follows:

\$0000	Console
\$0001	Printer (Not possible under the 2.0 version of the shell.)
\$0002	Disk file

Assembler Reference Manual

Possible Errors:

\$04 Invalid parameter count
\$53 Parameter out of range

Coding Example:

```
                DirectionGS drRec
                .
                .
                .
drRec           dc      i'2'
drDevice        dc      i'1'
drDirect        ds      2
```

look at standard out
direction

Error (\$0145)

Write Error

Forms:

LAB ErrorGS REC

Description:

When an Apple IIGS tool call returns an error, your program can use this function to print out the name of the tool and the appropriate error message. This function makes it unnecessary for your program to store a complete table of error messages for tool calls. The error number is placed in the accumulator by the tool; you need only store the accumulator value in the parameter block and execute this call to print the error message to standard error output.

Parameter List:

0	pCount
1	
2	error
3	

Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [1]
		Parameter count; must be 1.	
\$02-\$03	error	Error number	2-byte value [\$0000-\$FFFF]

This parameter specifies the error number returned by the tool call.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```

                                ErrorGS erRec
                                .
                                .
                                .
erRec    dc    i'1'
erError  ds    2
```

Execute (\$014D)

Execute Commands

Forms:

LAB ExecuteGS REC

Description:

This function sends a command or list of commands to the ORCA Shell command interpreter.

Parameter List:

0	pCount
1	
2	flag
3	
4	comm
5	

Offset	Label	Parameter Name	Size and Type [range of values]
--------	-------	----------------	---------------------------------

\$00-\$01	pCount	parameter count	2-byte value [2]
-----------	--------	-----------------	------------------

Parameter count; must be 2.

\$02-\$03	flag	Define new variables table	2-byte value [\$0000 or \$8000]
-----------	------	----------------------------	---------------------------------

If you set the most significant bit of this flag to 1 (binary), then a new variable table is not defined when the commands are executed. This flag is used to execute an Exec file with an EXECUTE command; if no new variable table is defined, then variables defined by the list of commands modify the current variable table. If this flag is set to \$0000, a new variable table is defined for the list of commands being executed; the current variable table is not modified. Exec files, variables, and the EXECUTE command are described in Chapter 12.

\$04-\$07	comm	Address of command string	4-byte pointer [\$0000 0000–\$00FF FFFF]
-----------	------	---------------------------	--

The address of the buffer in which you place the commands. If you include more than one command, separate the commands with semicolons (;) or carriage return characters (\$0D). Terminate the command string with a null character (\$00). Any output is sent to standard output.

Chapter 24: Shell Calls

The last command in the command stream must end with a carriage return character (\$0D).

If the variable {exit} is not null and any command returns a non-zero error code, then any remaining commands are ignored. Error codes and variables are described in Chapter 12.

Possible Errors:

\$04 Invalid parameter count
\$?? Any error returned from the last command or program executed by the list of commands executed.

Coding Example:

```
ExecuteGS exRec
.
.
.
exRec    dc    i'2'
         dc    i'0'                create a new variable table
         dc    a4'commands'        address of command list
commands dc    c'Catalog',i1'13,0'
```

ExpandDevices (\$0154)

Expand a Path Name

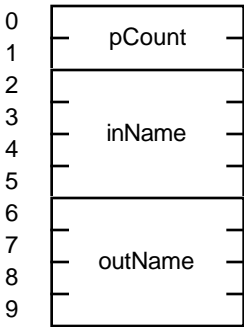
Forms:

LAB ExpandDevicesGS REC

Description:

This call is used to support prefix numbers, device names, and directory walking. Many GS/OS calls do not directly support the use of prefix numbers and device names like the shell uses, and none support directory walking (the .. operator). This call takes a path name as input. The input path name can contain prefix numbers (as in 17:HELP:CAT), device numbers (as in .d3:myfile) and the directory walking operator (as in ../prefix:myfile). This command also remaps the old ProDOS prefixes, 0 to 7, into the equivalent GS/OS prefixes. It also accepts full and partial path names which do not contain prefix numbers, device numbers, or the .. operator. In all cases, it returns a valid full GS/OS path name.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [2] Parameter count; must be 2.
\$02-\$05	inName	pointer to input path name	4-byte pointer [\$0000 0000–\$00FF FFFF] Pointer to the input name buffer. The input value is a GS/OS input string.
\$06-\$09	outName	pointer to output path name buffer	4-byte pointer [\$0000 0000–\$00FF FFFF] Pointer to the output name buffer. The output buffer is a GS/OS output string.

Chapter 24: Shell Calls

Possible Errors:

\$04 Invalid parameter count
\$53 Parameter out of range

Coding Example:

```
ExpandDevicesGS exRec
.
.
.
exRec    dc      i'2'
         dc      a4'inName'
         dc      a4'outName'

inName   dOSIn   '13:Login'
outName  dOSOut  100
```

original path name
expanded path name

Export (\$0156)

Export a Variable

Forms:

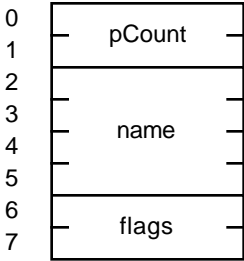
LAB ExportGS REC

Description:

The EXPORT call marks a shell variable as exportable. Marking {MyVariable} as exportable, for example, would be equivalent to executing the shell command

export MyVariable

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [2]
		Parameter count; must be 2.	
\$02-\$05	name	variable name	4-byte pointer [\$0000 0000–\$0FF FFFF]
		Pointer to the name of the variable to mark. The name is a length byte followed by the ASCII characters of the name.	
\$06-\$07	flags	variable's flags	2-byte value [\$0000 or \$0001]

The flags variable tells the shell whether the variable is to be marked as exportable or not. If flags is 1, the variable is marked as exportable. If flags is 0, the the variable is marked as not exportable.

Possible Errors:

- \$04 Invalid parameter count
- \$53 Parameter out of range

Chapter 24: Shell Calls

Coding Example:

```
ExportGS exRec
.
.
.
exRec    dc    i'2'
         dc    a4'name'
         dc    i'1'

inName   dOSIn  'myVariable'
```

FastFile (\$014E)

Fast File Handler

Forms:

LAB FastFileGS REC

Description:

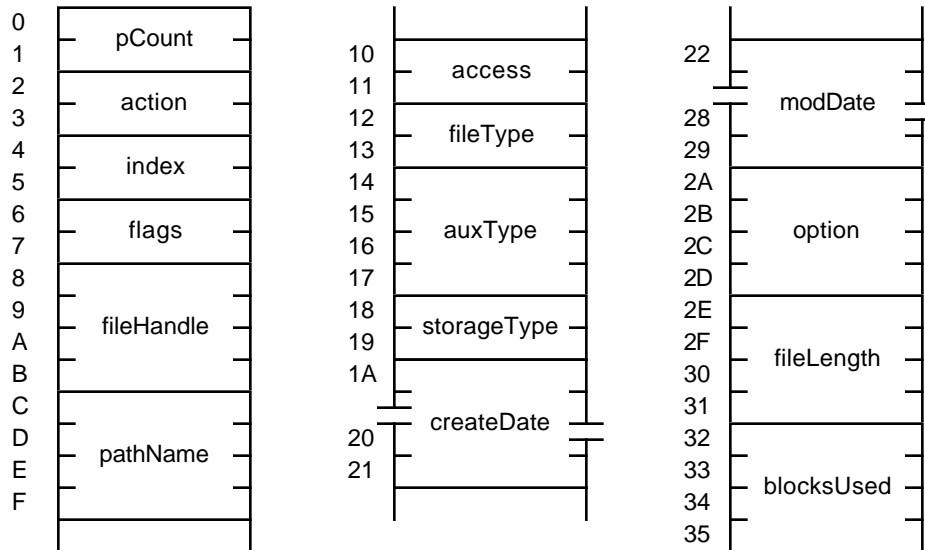
The FastFile call sets up a high-level disk caching system that can be used by any program running under the shell. There are two ways the system can be used. In the first, you use the FastFile call to load and save your files, being sure to purge them when you are finished with the files. Then, if any subsequent FastFile load is issued while the file is still in memory, the FastFile call simply locks the file and returns a pointer to it, saving a great deal of disk access time. The second method is intended solely for the use of compilers and linkers. Two action codes – 4 and 2 – allow you to add a file to the file list that is not on disk, and to read a file only from memory. This method is used by all ORCA languages and linkers from the desktop when you do a memory-based compile, and from the text shell when the +M flag is used. The object modules produced by the compiler are passed directly to the linker, with no disk access. Naturally, this speeds up the compile process.

Without some form of check, it would be possible for a program that did not use the FastFile call to change a file on disk, after which a program using the FastFile system might get an old copy of the file left around from a previous call to the FastFile system. For example, suppose you use the editor (which supports the FastFile system) to edit a file. Next, you use a utility that does not use the FastFile call to make some change to the file. Finally, you compile the file. The compiler should use the copy on disk, not the copy left from the editor. Two methods are used to make sure the FastFile call always returns the correct file. First, any GS/OS Open, Destroy, or SetFileInfo call will remove the file from the FastFile system. Second, any time the FastFile call finds a file in memory, it checks the disk copy (if any) to ensure that the modification date and time match. If they do not, the disk copy is used.

See the description of the action parameter, below, for a list of the actions the FastFile call can do for you.

Chapter 24: Shell Calls

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [5-14]

Parameter count. Any value from 5 to 14 is allowed.

\$02-\$03	action	action to take	2-byte value [\$00 – \$07]
-----------	--------	----------------	----------------------------

The action parameter tells the FastFile system which action to take.

Load (\$00): You must provide the path name and flags parameters. The file will be loaded and locked, and all of the parameters from fileHandle to blocksUsed will be filled in with the correct values.

Indexed Load (\$01): This call works like load, except that you provide an index rather than a path name. The FastFile call returns the file at that index, or a file not found error if there are no files for that index. For example, if there are three files in the FastFile system, you could catalog them by calling FastFile with indexes of 1, 2, 3 and 4. When the index is 4, an error is returned, indicating that you have reached the end of the list of files.

Load From Memory (\$02): This call works like Load, except that the file will only be returned if it is in the FastFile system. If the file exists on disk, but has not been loaded by another FastFile call, a file not found error is returned. This call is intended primarily for use by compilers and the linker to pass files without writing them to disk.

Save (\$03): You must fill in all of the fields except index and blocksUsed. The file is written to disk, but remains in memory. Use the Purge call to allow the Memory Manager to reclaim the memory if it is needed.

Add (\$04): You must fill in all of the fields except index and blocksUsed. The file is placed in the FastFile system, where other programs can access it, but it is not written to disk. You may use the Purge call to allow the Memory Manager to reclaim the memory, but this is usually not appropriate. The main use for this call is for a compiler to pass an object module to the linker without accessing the disk.

Delete (\$05): You must fill in the path name and action fields. The file is removed from the FastFile system (if it is there). The file is not deleted from the disk.

Remove (\$06): You must fill in the path name and action fields. As with the delete call (\$05), the file is removed from the FastFile list, but this call does not do a Memory Manager DisposeHandle call to remove the file itself from memory. This call is generally used by desktop shells to prevent other programs from seeing a file without disposing of the contents of the file.

Purge (\$07): The file is unlocked and marked purgeable with a purge level of 2. This allows the Memory Manager to reuse the memory if it is needed, but leaves the file in memory if the memory is not needed for some other purpose.

\$04–\$05	index	index number of file to load	2-byte value [\$0001 – \$7FFF]
-----------	-------	---------------------------------	--------------------------------

The index parameter is only used when the action code is \$01 (indexed load). It specifies the file to load by index number, starting with 1. See the description of indexed load, above, for details.

\$06–\$07	flags	how to handle file	2-byte value [\$0000 – \$FFFF]
-----------	-------	--------------------	--------------------------------

The flags field tells the FastFile system how to handle the file. If the most significant bit (\$8000) is set, the file exists on disk as well as in the FastFile system. This flag enables the check of the mod date/time, made each time the file is loaded. If the mod date/time for the disk copy of the file differs from the memory copy, the disk copy is used. The next most significant bit (\$4000) is set if the file may be purged. If this flag is clear, Purge calls (action \$07) are ignored. Normally, when you are using the FastFile system to load and save disk files, you would set this field to \$C000. For passing files through the FastFile system, as is done by the compiler and linker, the flags would normally be set to \$0000. The unused bits (the least significant 14 bits) should be set to 0.

Chapter 24: Shell Calls

\$08-\$0B	fileHandle	handle to file in memory	4-byte pointer [\$0000 0000-\$00FF FFFF]
-----------	------------	--------------------------	--

This is the handle of the file. It is returned by the load calls, and must be provided when saving the file or adding it to the file list.

\$0C-\$35	...	see diagram	see diagram
-----------	-----	-------------	-------------

The remaining parameters have the same format, use, and restrictions as the corresponding parameters for a GS/OS GetFileInfo or SetFileInfo call, with the GetFileInfo parameters applying to loads and the SetFileInfo parameters applying to stores. In fact, the FastFile call makes GetFileInfo and SetFileInfo calls with these parameters.

Possible Errors:

\$04	Invalid parameter count
\$53	Parameter out of range

Coding Example:

```
ExportGS exRec
.
.
.
exRec    dc    i'2'
         dc    a4'name'
         dc    i'1'

inName   dOSIn  'myVariable'

        lla    ffPathName,name          load the file
        stz    ffAction
        lda    #$C000
        sta    ffFlags
        FastFileGS ffRec
;
; Put code to use and change the file here. The save call below is
; only needed if you change the file. Since the load call filled in
; all of the fields, we do not have to change much to do the save.
;
        lla    ffPathName,name          save the file
        lda    #3
        sta    ffAction
        FastFileGS ffRec
        lla    ffPathName              purge the file
        lda    #7
        sta    ffAction
        FastFileGS ffRec
.
.
```

Assembler Reference Manual

ffRec	dc	i'14'	parameter count
ffAction	ds	2	action
ffIndex	ds	2	file index
ffFlags	ds	2	FastFile flags
ffFileHandle	ds	4	file handle
ffPathName	ds	4	path name
ffAccess	ds	2	GS/OS file access code
ffFileType	ds	2	GS/OS file type
ffAuxType	ds	4	GS/OS auxiliary file type
ffStorageType	ds	2	GS/OS storage type
ffCreateDate	ds	8	GS/OS create date/time
ffModDate	ds	8	GS/OS modification date/time
ffOption	dc	a4'0'	GS/OS file options
ffFileLength	ds	4	file length (GS/OS & shell)
ffBlocksUsed	ds	4	GS/OS block count
name	dOSIn	'myFile'	

GetCommand (\$015D)

Get a Command Table Entry

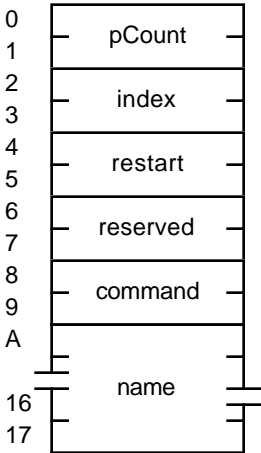
Forms:

LAB GetCommandGS REC

Description:

The GetCommand call returns the name, restart flag and command number for a command in the command table. The first command is accessed with an index number of 1, the next with an index of 2, and so forth. All commands are numbered sequentially internally. If an index is supplied for which there is no command, the length of the name is set to 0.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [5] Parameter count; must be 5.
\$02-\$03	index	index number of command	2-byte value [\$0001-\$7FFF] Number of the command to return. Commands are numbered sequentially, starting with number 1.
\$04-\$05	restart	restartability flag	2-byte value [\$0000-\$0001] 1 if the language or utility is restartable, 0 if it is not. This field is not used by built in commands.

Assembler Reference Manual

\$06-\$07	reserved	field not used	2-byte value [\$0000]
-----------	----------	----------------	-----------------------

Not used.

\$08-\$09	command	number assigned to command	2-byte value [\$0000-\$7FFF]
-----------	---------	----------------------------	------------------------------

Command number. Utilities have a command number of 0. Built-in commands have the number specified in the SYSCMND file. Languages return the language number ORed with \$8000.

\$0A-\$19	name	command's name	16-byte string
-----------	------	----------------	----------------

This is the name of the command, starting with a length byte. If there is no command for the index supplied, the length of the name is set to 0.

Unlike most strings returned from GS/OS or the shell, this name field is of a small, fixed length, so it is returned in a p-string style array, rather than in a GS/OS output buffer.

Possible Errors:

\$04	Invalid parameter count
------	-------------------------

Coding Example:

```
                                list all of the commands
lb1      stz      index
          GetCommandGS gcRec
          lda      gcName
          and      #$00FF
          beq      lb2
          puts     gcName-1,cr=t
          inc      index
          bra      lb1
lb2      anop
          .
          .
          .
gcRec    dc        i'5'
gcIndex  ds        2
gcRestart ds      2
          ds        2
gcCommand ds      2
gcName   ds       16
```


GetIODevices (\$015C)

Get a List of the IO Devices

Forms:

LAB GetIODevicesGS REC

Description:

This call is obsolete. It was used under older versions of the shell to handle I/O trapping inside of programs like the ORCA Desktop Development Environment. I/O trapping is now handled through the GS/OS console driver.

GetLang (\$0143)

Get Language Number

Forms:

LAB GetLangGS REC

Description:

This function reads the current language number. Language numbers are described in Chapter 12.

Parameter List:

0	pCount
1	
2	lang
3	

<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type [range of values]</u>
\$00-\$01	pCount	parameter count	2-byte value [1]
		Parameter count; must be 1.	
\$02-\$03	lang	Language number	2-byte result [\$0000-\$7FFF]

The current ORCA language number. The current language number is set by the ORCA editor when it opens an existing file, or by the user with a ORCA shell command.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```

                                GetLangGS glRec
                                .
                                .
                                .
glRec      dc      i'1'
glLang     ds      2
```

GetLInfo (\$0141)

Get Language Info

Forms:

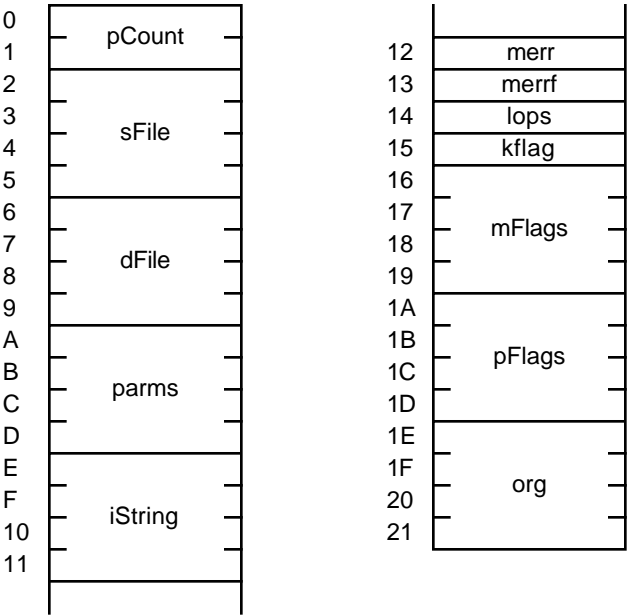
LAB GetLInfoGS REC

Description:

This function is used by an assembler, compiler, linker, or editor to read the parameters that are passed to it. When you make this call, you reserve the specified amount of space for each parameter in the parameter block; when the ORCA shell returns control to your program, you can then read the parameter block to obtain the information you need.

Use the SetLInfo call when your program is finished before executing an RTL or GS/OS Quit to return control to the shell.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [11]
		Parameter count; must be 11.	
\$02-\$05	sfile	Address of source file name	4-byte pointer [\$0000 0000–\$00FF FFFF]

The address of a GS/OS output buffer into which the shell will put the file name of the source file; that is, the file that the compiler or assembler is to process. The file name can be any valid GS/OS file name, and can be a partial or full path name.

For editors, this field can contain more than one file name. For multiple file names, the file names will be separated by a single space.

\$06–\$09	dfile	Address of output file name	4-byte pointer [\$0000 0000– \$00FF FFFF]
-----------	-------	--------------------------------	--

The address of a GS/OS output buffer into which the shell puts the file name of the output file (if any); that is, the file that the compiler or assembler writes to. The file name can be any valid GS/OS file name, and can be a partial or full path name.

\$0A–0D	parms	Address of parameter list	4-byte pointer [\$0000 0000– \$00FF FFFF]
---------	-------	------------------------------	--

The address of a GS/OS output buffer into which the shell puts the list of names from the NAMES= parameter list in the ORCA shell command that called the assembler or compiler. If there was no NAMES parameter list, the length of the buffer will be set to 0.

\$0E–\$11	istring	Address of input strings	4-byte result [\$0000 0000– \$00FF FFFF]
-----------	---------	-----------------------------	---

The address of a GS/OS output buffer into which the shell puts the string of commands to be passed on to a specific language compiler. For example, if the COMPILE command includes the parameter CC=(-I/CINCLUDES/), then the string enclosed in parentheses is found in that buffer when the C compiler is called.

\$12	merr	Maximum error level allowed	1-byte result [\$00–\$10]
------	------	--------------------------------	---------------------------

If the maximum error level found by the assembler, compiler, or linker (merrf) is greater than merr the ORCA shell does not call the next program in the processing sequence. For example, if you use the ASML command to assemble and link a program, but the assembler finds an error level of 8 when merr equals 2, then the linker is not called when the assembly is complete.

\$13	merrf	Maximum error level found	1-byte result [\$00–\$FF]
------	-------	------------------------------	---------------------------

This field is used by the SetLInfo call to return the maximum error level found. This field contains the error level returned by the last

Chapter 24: Shell Calls

compiler in a multi-language compile. If this is the first compile, this field is \$00.

\$14 lops Operations flags 1-byte result [\$00–\$10]

This field is used to keep track of the operations that are to be performed by the system. The format of this byte is as follows:

Bit:	7	6	5	4	3	2	1	0
Value:	0	0	0	0	0	E	L	C

where: C = Compile
L = Link
E = Execute

When a bit is set (1), the indicated operation is to be done. For example, the COMPILE command sets bit 0, while the CMLPG command sets bits 0, 1, and 2. When a compiler finishes its operation and returns control to the ORCA shell, it clears bit 0 unless a file with another language is appended to the source.

\$15 kflag Keep flag 1-byte result [\$00–\$03]

This flag indicates what should be done with the output of a compiler, assembler, or linker, as follows:

<u>Kflag</u>	
<u>Value</u>	<u>Meaning</u>

\$00	Do not save output.
------	---------------------

\$01	Save to an object file with the root file name pointed to by dfile. For example, if the output file name pointed to by dfile is PROG, then the first segment to be executed should be put in PROG.ROOT and the remaining segments should be put in PROG.A. For linkers, save to a load file with the name pointed to by dfile (for example, PROG).
------	--

\$02 The .ROOT file has already been created (by another language compiler, for example). In this case, the first file created by the compiler or assembler should end in the .A extension.

\$O3 At least one alphabetic suffix has already been used. In this case, the compiler or assembler must search the directory for the highest alphabetic suffix that has been used, and then use the next one. For example, if PROG.ROOT, PROG.A, and PROG.B already exist, the compiler should put its output in PROG.C.

\$16-\$19	mflags	Flags with a minus sign	4-byte result [binary string]
-----------	--------	-------------------------	-------------------------------

This parameter passes command-line-option flags such as `-L` or `-C`. The first 26 bits of these four bytes represent the letters A–Z, arranged with A as the most significant bit of the most significant byte; the bytes are ordered least significant byte first. The bit map is as follows:

```
11000000 11111111 11111111 11111111
YZ        QRSTUVWX IJKLMNOP ABCDEFGH
```

For each flag set with a minus sign in the command, the corresponding bit in this parameter is set to 1. See the discussions of the `LINK` and `ASML` commands in Chapter 12 for descriptions of these option flags.

\$1A–\$1D	pflags	Flags with a plus sign	4-byte result [binary string]
-----------	--------	------------------------	-------------------------------

This parameter passes command-line-option flags such as +L or +C. The first 26 bits of these four bytes represent the letters A–Z; the bit map for this parameter is the same as for the mflags parameter. See the discussions of the ASSEMBLE and LINK commands in Chapter 12 for descriptions of these option flags.

\$1E-\$21	org	Origin	4-byte result [\$0000 0000-\$FFFF FFFF]
-----------	-----	--------	---

This field is used on entry to an editor to provide a displacement into the file. The editor can then place at the top of the screen the line that corresponds to this displacement.

Possible Errors:

\$04 Invalid parameter count

Chapter 24: Shell Calls

Coding Example:

```
                GetLInfo glRec
                .
                .
                .
glRec           dc      i'11'
glSFile         dc      a4'inFile'
glDFile         dc      a4'outFile'
glParms         dc      a4'parmList'
glIString       dc      a4'cmdList'
glMerr          ds      1
glMerrf         ds      1
glLops          ds      1
glKFlag         ds      1
glMFlags        ds      4
glPFlags        ds      4
glOrg           ds      4

inFile          dOSOut  256
outFile         dOSOut  256
parmList        dOSOut  256
cmdList         dOSOut  256
```

InitWildcard (\$0149)

Initialize Wildcards

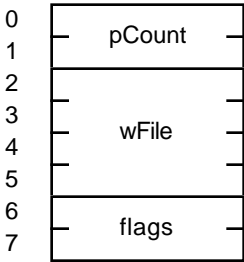
Forms:

LAB InitWildcardGS REC

Description:

This function provides to the ORCA shell a file name that can include a wildcard character. The shell can then search for file names matching the file name you specified when it receives a NextWildcard command. This function accepts any file name, whether it includes a wildcard or not, and expands device names (such as .D1), prefix numbers, and the double-period (..) before the file name is passed on to GS/OS. Therefore, you should call this function every time you want to search for a file name. Doing so will assure that your routine supports all of the conventions for partial path names that the user expects from ORCA.

Parameter List:



<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type [range of values]</u>
\$00-\$01	pCount	parameter count	2-byte value [2]
\$02-\$05	wFile	Address of path name	4-byte pointer [\$0000 0000–\$00FF FFFF]

Parameter count; must be 2.

The address of a buffer containing a path name or partial path name that can include a wildcard character. Examples of such path names are:

A=
:ORCA:MYPROGS:?.ROOT
.D2:HELLO

When you execute a NextWildcard call, the shell finds the next file name that matches the file name pointed to by wFile. If the wild card character you specified was a question mark (?), then the file name is written to standard output and you are prompted for

Chapter 24: Shell Calls

confirmation before the file is acted on or the next file name is found. The use of wildcard characters is described in the section “Wildcards” in Chapter 12.

\$06–\$07 flags Prompting flags. 2-byte value

This field is a series of bit flags that tell the wildcard handler to process the matching files in certain ways.

If bit \$8000 is set, prompting is not allowed; that is, a question mark (?) is treated as if it were an equal sign (=).

If bit \$4000 is set and prompting is being used, only the first choice accepted by the user (that is, the first choice for which the user types a Y in response to the prompt) is acted on. The second flag is for use with commands that can act on only one file, such as RENAME or EDIT.

If bit \$2000 is set, and if the wildcard handler returns a directory, it will also return all of the files in the directory. The first file returned will be the directory itself, followed by the first file in the directory, and so on. If the directory contains other directories, they are expanded, too.

If bit \$1000 is set and bit \$2000 is also set, the wildcard handler returns the files that are in a directory before it returns the directory itself.

All other flags are reserved, and should be set to 0.

Possible Errors:

\$04 Invalid parameter count

\$?? Errors for the following GS/OS and Memory Manager calls. See the *GS/OS™ Reference Manual* and the *Apple Toolbox Reference* manuals for descriptions of these errors.

Open
Read
Close
Dispose
GetFileInfo
GetEOF
Lock
NewHandle

Assembler Reference Manual

Coding Example:

```
                InitWildcardGS inRec
                .
                .
                .
inRec           dc      i'2'
inWFile         dc      a4'path'
inFlags         dc      i'0'

path            dOSIn    '=.macros'
```

NextWildcard (\$014A)

Next Wildcard Name

Forms:

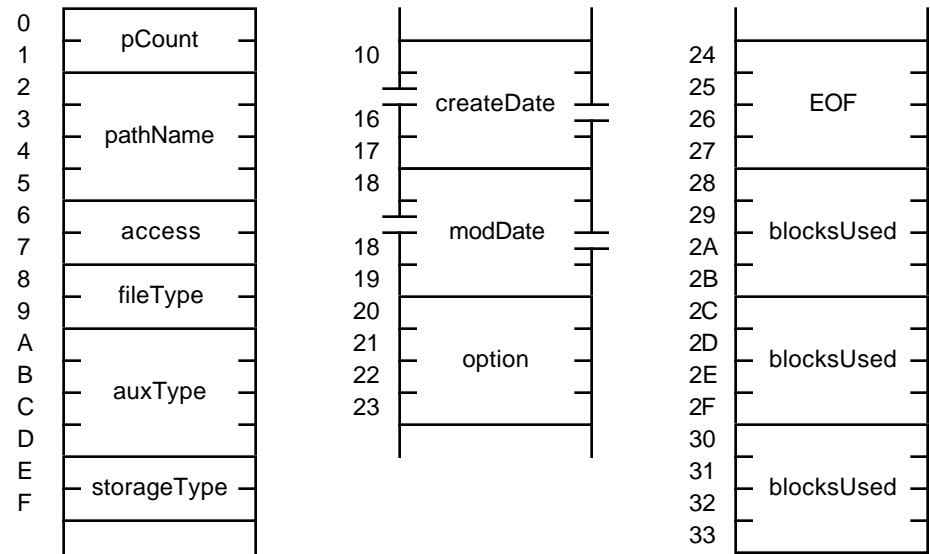
LAB NextWildcardGS REC

Description:

Once a file name that includes a wildcard has been supplied to the shell with an InitWildcard call, the NextWildcard call causes the shell to find the next file name that matches the wildcard file name. For example, if the wildcard file name specified in InitWildcard is :ORCA:UTILITY:XREF.?, then the first file name returned by the shell in response to a NextWildcard call might be :ORCA:UTILITY:XREF.ASM.

The parameter list, range of values, and the meanings of the parameters correspond exactly to the parameter list for the GS/OS GetFileInfo call.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [1-12]
		Parameter count; and value from 1 to 12.	
\$02-\$33	...	see diagram	see diagram

The remaining NextWildcard parameters correspond exactly to the parameters returned by the GS/OS GetFileInfo call.

Assembler Reference Manual

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```
                                NextWildcardGS nwRec
                                .
                                .
                                .
nwRec      dc      i'1'
nwWFile    dc      a4'path'

path       dosOut 256
```

PopVariables (\$0157)

Pop the Previous Variable List

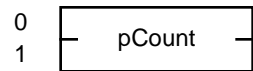
Forms:

LAB PopVariablesGS REC

Description:

PopVariables restores the shell variable list to the state that it was in when the last PushVariables call was made. If no PushVariables call has been made, or if a PopVariables call has been made for each PushVariables call, the call is ignored.

Parameter List:



<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type [range of values]</u>
\$00-\$01	pCount	parameter count	2-byte value [0]

Parameter count; must be 0.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```
PopVariablesGS pvRec
.
.
.
pvRec      dc      i'0'
```

PushVariables (\$0158)

Push the Variable List

Forms:

LAB PushVariablesGS REC

Description:

PushVariables saves the current state of the shell variable list, then sets up a new shell variables list using the rules for creating a new EXEC file; i.e., a new list of variables is created, and any exportable variables are copied into the new list. PopVariables can be called to destroy the variable list, returning to the one pushed.

Parameter List:

0	<div><div></div><div>pCount</div><div></div></div>
1	

<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type [range of values]</u>
\$00-\$01	pCount	parameter count	2-byte value [0]

Parameter count; must be 0.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```

                PushVariablesGS pvRec
                .
                .
                .
pvRec          dc      i'0'
```

ReadIndexed (\$0148)

Indexed Shell Variable Read

Forms:

LAB ReadIndexedGS REC

Description:

You can use this function to read the contents of the variable table for the command level at which the call is made. To read the entire contents of the variable table, you must repeat this call, incrementing the index number by 1 each time, until the entire contents have been returned.

Parameter List:

0	pCount
1	
2	name
3	
4	
5	value
6	
7	
8	index
9	
A	
B	export
C	
D	

<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type [range of values]</u>
\$00-\$01	pCount	parameter count	2-byte value [4]

Parameter count; must be 4.

\$02-\$05	name	Pointer to name of variable	4-byte pointer [\$0000 0000–\$00FF FFFF]
-----------	------	-----------------------------	--

This is a pointer to a GS/OS output buffer in which the shell places the name of the next variable in the variable table. A null string is returned when the index number exceeds the number of variables in the variable table.

In the 2.0 version of the shell, the longest possible variable name is 256 characters long.

Assembler Reference Manual

\$06–\$09	value	Pointer to value of variable	4-byte pointer [\$0000 0000–\$00FF FFFF]
-----------	-------	------------------------------	--

This is a pointer to a GS/OS output buffer into which the shell places the value of the variable. The value consists of a null string (that is, the length byte is \$00) for an undefined variable.

Variable values can be up to 65531 characters long.

\$0A–\$0B	index	Index number	2-byte value [\$0000–\$FFFF]
-----------	-------	--------------	------------------------------

This is an index number that you provide. Start with \$01 and increment the number by 1 with each successive ReadIndexed call until there are no more values in the variable table.

\$0C–\$0D	export	Export flag	2-byte value [\$0000–\$FFFF]
-----------	--------	-------------	------------------------------

The shell returns this value, which will be 0 if the variable is not marked as exportable, and non-zero if the variable is marked as exportable.

Possible Errors:

\$04 Invalid parameter count

\$?? Errors for the LOCK and UNLOCK Memory Manager calls. See the *Apple Toolbox Reference* manual for descriptions of these errors.

Coding Example:

```
        lda    #1
        sta    riIndex
ReadIndexedGS riRec
        .
        .
        .
riRec   dc      i'4'
riName  dc      a4'name'
riValue dc      a4'value'
riIndex ds      2
riExport ds      2

name     dOSOut 256
value    dOSOut 1024
```


ReadVariable (\$014B)

Read Shell Variable

Forms:

LAB ReadVariableGS REC

Description:

This function reads the string associated with a variable (that is, the value of the variable). The value returned is the one valid for the currently-executing Exec file and any Exec files called from that file, or for the interactive command interpreter and all Exec files called from the command interpreter (if that is the command level in use). Variables and Exec files are described in Chapter 12. Use the Set call to set the value of a variable.

Parameter List:

0	pCount
1	
2	name
3	
4	
5	value
6	
7	
8	
9	export
A	
B	

<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type [range of values]</u>
\$00-\$01	pCount	parameter count	2-byte value [3]
		Parameter count; must be 3.	
\$02-\$05	name	Pointer to name of variable	4-byte pointer [\$0000 0000–\$00FF FFFF]

This is a pointer to a GS/OS input buffer that contains the name of the variable whose value you wish to read.

Assembler Reference Manual

\$06–\$09	value	Pointer to value of variable	4-byte pointer [\$0000 0000– \$00FF FFFF]
-----------	-------	---------------------------------	--

This is a pointer to a OS/OS output buffer into which the shell places the value of the variable. The value consists of a null string (that is, the length byte is \$00) for an undefined variable.

\$0A–\$0B	export	Export flag	2-byte value [\$0000–\$FFFF]
-----------	--------	-------------	------------------------------

The shell returns this value, which will be 0 if the variable is not marked as exportable, and non-zero if the variable is marked as exportable.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```
ReadVariableGS rvRec
.
.
.
rvRec      dc      i'3'
rvName     dc      a4'name'
rvValue    dc      a4'value'
rvExport   ds      2

name       dOSIn   'Status'
value      dOSOut  10
```

Redirect (\$0150)

Redirect I/O

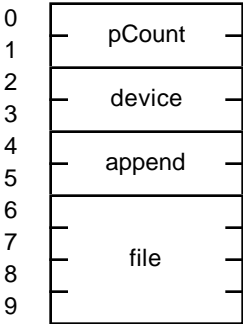
Forms:

LAB RedirectGS REC

Description:

This function instructs the shell to redirect input or output to the console, or a disk file.

Parameter List:



<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type [range of values]</u>
---------------	--------------	-----------------------	--

\$00-\$01	pCount	parameter count	2-byte value [3]
-----------	--------	-----------------	------------------

Parameter count; must be 3.

\$02-\$03	device	Device number	2-byte value [\$0000-\$0002]
-----------	--------	---------------	------------------------------

This parameter indicates which type of input or output you wish to redirect, as follows:

- | | |
|--------|-----------------|
| \$0000 | Standard input |
| \$0001 | Standard output |
| \$0002 | Error output |

\$04-\$05	append	Append flag	2-byte value [\$0000-\$FFFF]
-----------	--------	-------------	------------------------------

This flag indicates whether redirected output should be appended to an existing file with the same file name, or the existing file should be deleted first. If append is 0, the file is deleted, if it is any other value, the output is appended to the file.

Assembler Reference Manual

\$06-\$09	file	Address of file name	4-byte pointer [\$0000 0000–\$00FF FFFF]
-----------	------	----------------------	--

The address of a GS/OS input buffer containing the file name of the file to or from which output is to be redirected. The file name can be any valid GS/OS file name, a partial or full path name, or the name of a character device.

Possible Errors:

\$04	Invalid parameter count
\$53	Parameter out of range
\$??	Errors for the following GS/OS calls. See the <i>GS/OS™ Reference Manual</i> and the <i>Apple Toolbox Reference</i> manuals for descriptions of these errors.

- Open
- Close
- Read
- Write
- Get End of File

Coding Example:

```

RedirectGS rdRec                                send error out to a printer
.
.
.
rdRec      dc      i'3'
rdDevice   dc      i'2'
rdAppend   dc      i'0'
rdFile     dc      a4'name'

name       dOSIn   '.Printer'
```

Set (\$0146) Set Shell Variable

Forms:

LAB SetGS REC

Description:

This function sets the value of a variable. If the variable has not been previously defined, this function defines it. Variables are described in Chapter 10. Use the ReadVariable call to read the current value of a variable and the ReadIndexed call to read a variable table.

Parameter List:

0	pCount	
1		
2	name	
3		
4		
5	value	
6		
7		
8		
9	export	
A		
B		

Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [3]
		Parameter count; must be 3.	
\$02-\$05	name	Pointer to name of variable	4-byte pointer [\$0000 0000–\$00FF FFFF]

This is a pointer to a GS/OS Input string in which you place the name of the variable whose value you wish to change. The name is an ASCII string. Only the first 255 characters of the name are significant.

Assembler Reference Manual

\$06–\$09	value	Pointer to value of variable	4-byte pointer [\$0000 0000–\$00FF FFFF]
-----------	-------	------------------------------	--

This is a pointer to a buffer in which you place the value to which the variable is to be set. The value is an ASCII string.

\$0A–\$0B	export	Export flag	2-byte value [\$0000–\$FFFF]
-----------	--------	-------------	------------------------------

Export flag. Set this parameter to 1 if the variable is exportable, and 0 if the variable cannot be exported.

Possible Errors:

\$04 Invalid parameter count

\$?? Errors for the following Memory Manager calls. See the *Apple II GS Toolbox Reference* manual for descriptions of these errors.

HLock
HUnlock
GrowHandle
SetHandleSize

Coding Example:

```
SetGS    svRec
.
.
.
svRec    dc    i'3'
svName   dc    a4'name'
svValue  dc    a4'value'
svExport dc    i'1'

name     dOSIn  'Prompt'
value    dOSIn  'My GS: '
```

SetIODevices (\$015B)

Set the IO Devices

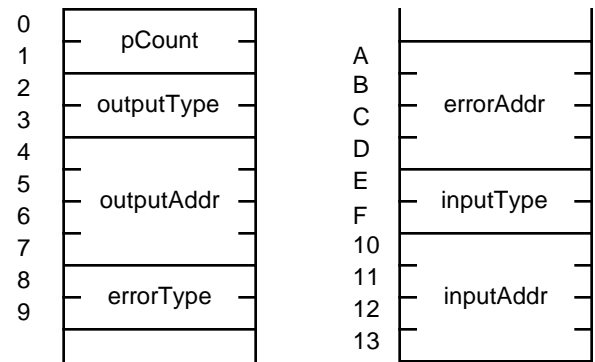
Forms:

LAB SetIODevicesGS REC

Description:

This call sets the device numbers and procedure addresses that will be used by the shell when output is sent to or read from .console. This is generally used by desktop development environments that want to trap the standard console output for display in a window.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [6] Parameter count; must be 6.
\$02-\$03	outputType	output device type	2-byte value [\$0000-\$0002] Used as the deviceType parameter of the Text Toolkit's SetOutputDevice call when the shell changes standard out to .CONSOLE.
\$04-\$07	outputAddr	output slot number or pointer to driver	4-byte value [\$0000 0000-\$0FF FFFF] Used as the ptrOrSlot parameter of the Text Tools' SetOutputDevice call when the shell changes standard out to .CONSOLE.

Assembler Reference Manual

\$08–\$09	errorType	error device type	2-byte value [\$0000–\$0002] Used as the deviceType parameter of the Text Tools' SetErrorDevice call when the shell changes error out to .CONSOLE.
\$0A–\$0D	errorAddr	error slot number or pointer to driver	4-byte value [\$0000 0000– \$0FF FFFF] Used as the ptrOrSlot parameter of the Text Tools' SetErrorDevice call when the shell changes error out to .CONSOLE.
\$0E–\$0F	inputType	input device type	2-byte value [\$0000–\$0002] Used as the deviceType parameter of the Text Toolkit's SetInputDevice call when the shell changes standard in to .CONSOLE.
\$10–\$13	inputAddr	input slot number or pointer to driver	4-byte value [\$0000 0000– \$0FF FFFF] Used as the ptrOrSlot parameter of the Text Toolkit's SetInputDevice call when the shell changes standard in to .CONSOLE.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```
                SetIODevicesGS siRec                use text shell defaults
                .
                .
                .
siRec           dc      i'6'
                dc      i'-1',a4'0'
                dc      i'-1',a4'0'
                dc      i'-1',a4'0'
```


SetLang (\$0144)

Set Language Number

Forms:

LAB SetLangGS REC

Description:

This function sets the current language number. Language numbers are described in the section “Command Types” in Chapter 12.

Parameter List:

0	pCount
1	
2	lang
3	

Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [1]
		Parameter count; must be 1.	
\$02-\$03	lang	Language number	2-byte value [\$0000–\$7FFF]

The ORCA language number to which the current ORCA language should be set. If the language specified is not installed (that is, not listed in the command table), then the “language not available” error is returned.

Possible Errors:

- \$04 Invalid parameter count
- \$80 Language not available

Coding Example:

```
SetLangGS slRec
.
.
.
slRec    dc    i'1'
         dc    i'4'
```

SetLInfo (\$0142)

Set Language Info

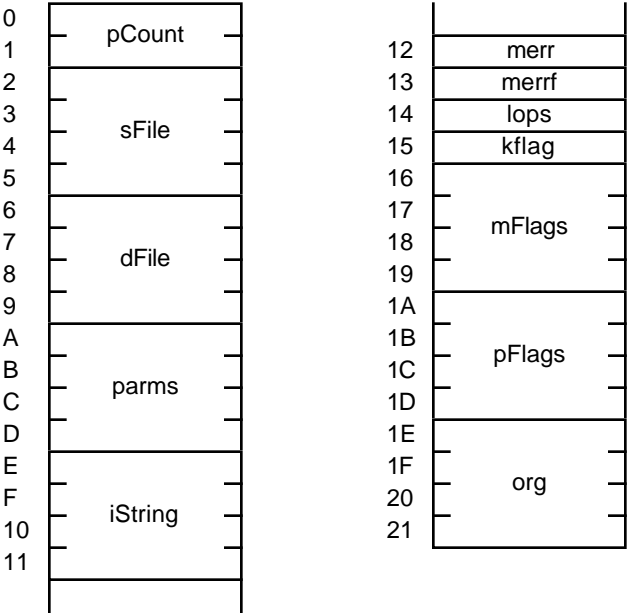
Forms:

LAB SetLInfoGS REC

Description:

This function is used by an assembler, compiler, linker, or editor to pass parameters to the ORCA shell before executing an RTL to return control to the shell. It can also be used by a shell program under which you are running ORCA to pass parameters to the ORCA shell.
Use the GetLInfo call to read parameters passed to your assembler, compiler, linker, or editor.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [11]
		Parameter count; must be 11.	
\$02-\$05	sfile	Address of source file name	4-byte pointer [\$0000 0000-\$00FF FFFF]
		The address of a GS/OS input string containing the file name of the source file; that is, the next file that a compiler or assembler is	

Chapter 24: Shell Calls

to process. The file name can be any valid GS/OS file name, and can be a partial or full path name.

\$06–\$09	dfile	Address of output file name	4-byte pointer [\$0000 0000–\$00FF FFFF]
-----------	-------	-----------------------------	--

The address of a GS/OS input string containing the file name of the output file (if any); that is, the file that the compiler or assembler writes to. The file name can be any valid GS/OS file name, and can be a partial or full path name.

\$0A–0D	parms	Address of parameter list	4-byte pointer [\$0000 0000–\$00FF FFFF]
---------	-------	---------------------------	--

The address of a GS/OS input string containing the list of names from the NAMES= parameter list in the ORCA shell command that called the assembler or compiler. The compiler can remove or modify these names as it processes them, so this list can be different from the one received through the GetLInfo call.

\$0E–\$11	istring	Address of input strings	4-byte pointer [\$0000 0000–\$00FF FFFF]
-----------	---------	--------------------------	--

A placeholder for the address of a GS/OS input string containing the string of commands passed to the compiler. This command string is not reused by the shell, so it is not necessary to pass it back to the shell with the SetLInfo call.

\$12	merr	Maximum error level allowed	1-byte value [\$00–\$10]
------	------	-----------------------------	--------------------------

If the maximum error level found by the assembler, compiler, or linker is greater than merr, then the shell does not call the next program in the processing sequence. For example, if you use the ASML command to assemble and link a program, but the assembler finds an error level of 8 when merr equals 2, then the linker is not called when the assembly is complete.

\$13	merrf	Maximum error level found	1-byte value [\$00–\$FF]
------	-------	---------------------------	--------------------------

This field is used by the SetLInfo call to return the maximum error level found. If merrf is greater than merr, then no further processing is done by the shell. If the high bit of merrf is set, then merrf is considered to be negative; a negative value of merrf indicates a fatal error (normally, all fatal errors are flagged as merrf=\$FF). In this case, processing terminates immediately and control is passed by the shell to the ORCA editor. See also the discussion of the org parameter.

\$14 lops Operations flags 1-byte value [\$00–\$10]

This field is used to keep track of the operations that have been performed by the system. The format of this byte is as follows:

Bit:	7	6	5	4	3	2	1	0
Value:	0	0	0	0	0	E	L	C

where C = Compile
 L = Link
 E = Execute

When a bit is set (1), the indicated operation is to be done. When a compiler finishes its operation and returns control to the shell, it clears bit 0 unless a file with another language is appended to the source. When a linker returns control to the shell, it clears bit 1. When you execute the linker with a LINK command, the linker clears bits 0 and 1.

\$15 kflag Keep flag 1-byte value [\$00–\$03]

This flag indicates what should be done with the output of a compiler, assembler, or linker, as follows:

<u>Kflag</u>	
<u>Value</u>	<u>Meaning</u>

\$00	Do not save output.
------	---------------------

\$01	Save to an object file with the root file name pointed to by dfile. For example, if the output file name pointed to by dfile is PROG, then the first segment to be executed should be put in PROG.ROOT and the remaining segments should be put in PROG.A. For linkers, save to a load file with the name pointed to by dfile (for example, PROG). A compiler or assembler will never set kflag to \$01, but a shell program calling ORCA might use this value.
------	---

Chapter 24: Shell Calls

\$02 The .ROOT file has already been created. In this case, the first file created by the next compiler or assembler should end in the .A extension.

\$03 At least one alphabetic suffix has been used. In this case, the compiler or assembler must search the directory for the highest alphabetic suffix that has been used, and then use the next one. For example, if PROG.ROOT, PROG.A, and PROG.B already exist, the compiler should put its output in PROG.C.

When the compiler or assembler passes control back to the shell, it should reset kflag to indicate which object files it has written; for example, if it found only one segment and created a .ROOT file but no .A file, then kflag should be \$02 in the SetLInfo call. See chapters 12, 14, and 17 for more information on object-file naming conventions.

\$16–\$19 mflags Flags with a minus sign 4-byte result [binary string]

This parameter passes command-line-option flags such as –L or –C. The first 26 bits of these four bytes represent the letters A–Z, arranged with A as the most significant bit of the most significant byte; the bytes are ordered least significant byte first. The bit map is as follows:

```
11000000 11111111 11111111 11111111
YZ       QRSTUVWX IJKLMNOP ABCDEFGH
```

For each flag set with a minus sign in the command, the corresponding bit in this parameter is set to 1. See the discussion of the ASML command in Chapter 12 for descriptions of these option flags.

\$1A–\$1D pflags Flags with a plus sign 4-byte result [binary string]

This parameter passes command-line-option flags such as +L or +C. The first 26 bits of these four bytes represent the letters A–Z; the bit map for this parameter is the same as for the mflags parameter. See the discussion of the ASML command in Chapter 12 for descriptions of these option flags.

\$1E–\$21 org Origin 4-byte value [\$0000 0000–\$FFFF FFFF]

The start address of the load file. The origin is used only by the linker. When a compile or assembly terminates with a fatal error

Assembler Reference Manual

(merrf=\$FF), the compiler or editor should put the displacement of the line containing the error into the org field. The editor can then place that line at the top of the screen.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```
                SetLInfo slRec
                .
                .
                .
slRec           dc      i'11'
slSFile         dc      a4'inFile'
slDFile         dc      a4'outFile'
slParms         dc      a4'parmList'
slIString       dc      a4'cmdList'
slMerr          dc      i'0'
slMerrf         dc      i'0'
slLops          dc      i'$07'
slKFlag         dc      i'2'
slMFlags        dc      i4'0'
slPFlags        dc      i4'0'
slOrg           dc      i4'0'

inFile          dOSIn   'source'
outFile         dOSIn   'keep'
parmList        dc      i'0'
cmdList         dc      i'0'
```

SetStopFlag (\$0159)

Set the Stop Flag

Forms:

LAB SetStopFlagGS REC

Description:

Set the value of the stop flag. A subsequent call to the Stop shell call will return this value.

Parameter List:

0	pCount
1	
2	flag
3	

Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [1] Parameter count; must be 1.
\$02-\$03	flag	stop flag	2-byte result [\$0000–\$0001] Use 1 to set the stop flag, and 0 to clear it.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```
                SetStopFlagGS ssRec
                .
                .
                .
ssRec          dc      i'1'
ssFlag         dc      i'1'
```

StopGS (\$0153)

Stop Processing?

Forms:

LAB StopGS REC

Description:

This function lets your application detect a request for an early termination of the program. The stop flag is set when the keyboard buffer is read after the user presses ⌘. (open apple period).

Parameter List:

0	pCount
1	
2	flag
3	

<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type [range of values]</u>
\$00-\$01	pCount	parameter count	2-byte value [1]
		Parameter count; must be 1.	
\$02-\$03	flag	Stop flag	2-byte result [\$0000–\$0001]

This flag is set (\$0001) by the shell when it finds a ⌘. in the keyboard buffer. When an ORCA utility reads from the keyboard as standard input, the shell reads the keyboard buffer and passes the keys on to the utility. When standard input is not from the keyboard, the shell still checks the keyboard buffer for ⌘. whenever a Stop call is executed. The flag is cleared (\$0000) when the Stop call is executed, when the utility program is terminated, or when windows are switched so that the utility program is no longer active.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```
                StopGS ssRec
                .
                .
ssRec           dc      i'1'
ssFlag          ds      2
```


UnsetVariable (\$0155)

Delete a Shell Variable

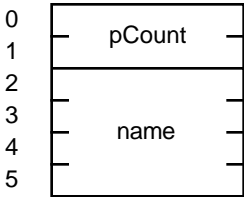
Forms:

LAB UnsetVariableGS REC

Description:

If the named shell variable exists, it is removed from the current list of variables. Subsequent attempts to read the value of the variable will always return the null string. This call is completely equivalent to using the shell command UNSET.

Parameter List:



<u>Offset</u>	<u>Label</u>	<u>Parameter Name</u>	<u>Size and Type [range of values]</u>
\$00-\$01	pCount	parameter count	2-byte value [1]
		Parameter count; must be 1.	
\$02-\$05	name	pointer to variable's name	4-byte pointer [\$0000 0000–\$00FF FFFF]
		Pointer to the name of the variable to unset. The name is a GS/OS input string.	

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```
UnsetVariableGS uvRec
.
.
.
uvRec    dc    i'1'
         dc    a4'name'

name    dosIn 'Exit'
```

Version (\$0147)

Shell Version

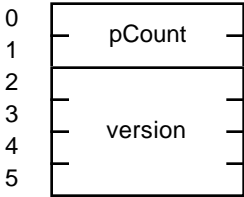
Forms:

LAB VersionGS REC

Description:

This function returns the version number of the ORCA shell that you are using.

Parameter List:



Offset	Label	Parameter Name	Size and Type [range of values]
\$00-\$01	pCount	parameter count	2-byte value [1]
		Parameter count; must be 1.	
\$02-\$05	version	Version number	4-byte result [\$0000 0000–\$3939 3939]

A four-byte ASCII string specifying the version number of the ORCA shell that you are using. The initial release returns 20 followed by two space characters, with the 1 in byte 0 (\$2020 3032), to indicate version number 2.0.

Possible Errors:

\$04 Invalid parameter count

Coding Example:

```

                VersionGS vrRec
                .
                .
                .
vrRec          dc      i'1'
vrVersion      ds      4
```

Chapter 25

Miscellaneous Macros

This section describes all of the macros that did not fit into one of the previous groups. The macros in this section are contained in the file M16.ORCA, located in the ORCAInclude directory in your libraries folder. You can access this file using the path name 13:ORCAInclude:m16.ORCA from your hard disk, or if you are using floppy disks, from :ORCA.Extras:Libraries:ORCAInclude:m16.ORCA.

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 once. 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
```

BGT

Branch on Greater Than

Forms:

LAB BGT BP

Operands:

LAB - Label.

BP - Branch point.

Description:

The 65816 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 a normal relative branch instructions. All registers are returned intact.

Coding Example:

```
                BGT     THERE
```

BLE

Branch on Less Than or Equal

Forms:

LAB BLE BP

Operands:

LAB - Label.

BP - Branch point.

Description:

The 65816 does not include an instruction to branch after a comparison on the condition that the register was less than or equal to the memory location. This macro serves the purpose, working just like a normal relative branch instructions. All registers are returned intact.

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. The contents of the accumulator and the X register are destroyed.

Coding Examples:

BUTTON	#0	branch to PRESSED if
BMI	PRESSED	button 0 is being pressed
BUTTON	NUM,RES	RES is result of examining button NUM

CLSUB**Local C Subroutine With Parameters, Stack Frame**

Forms:

```
LAB    CLSUB    [PARMS] ,SIZE
```

Operands:

LAB - Label.

PARMS - Parameter list.

SIZE - Size of direct page space.

Description:

See the description of the SUB macro.

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 or indirect 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. 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. If an overflow occurs during the conversion process, the V flag is set; otherwise it is cleared. The contents of all registers are lost.

Coding Examples:

```

!          CNV24  INT1,INT2    converts the 2-byte integer at INT1 into
!                                     a 4-byte integer, saving the result at
!                                     INT2
!          CNV42  {P1}        converts the number pointed to by P1
!                                     from a 4-byte integer to a 2-byte
!                                     integer
!          CNV2S  N1,STR       converts 2-byte integer N1 to a
!                                     string, saving at STR

```

CSUB C Subroutine With Parameters, Stack Frame

Forms:

LAB CSUB [PARMS], SIZE

Operands:

LAB - Label.

PARMS - Parameter list.

SIZE - Size of direct page space.

Description:

See the description of the SUB macro.

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. No registers are affected, other than the one specified.

Coding Example:

	DBNE	X, LAB	decrement X, branch to LAB if X is not 0
	DBEQ	Y, THERE	decrement Y, branch to THERE if Y is 0
	DBPL	NUM, TOP	decrement NUM, branch to TOP if NUM is
!			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 contents of the accumulator are lost.

Coding Example:

DEC2 NUM1

DEC4

Four-Byte Decrement

Forms:

LAB DEC4 N1

Operands:

LAB - Label.

N1 - Four-byte number to decrement.

Description:

N1 is decreased by one. The accumulator is lost.

Coding Example:

DEC2 NUM1

DOSIN

Define GS/OS Input String

Forms:

LAB DOSIN STRING

Operands:

LAB - Label.

STRING - String to define.

Description:

DW defines a GS/OS input string. That is, a count word followed by the ASCII characters which comprise the string. The count word specifies the number of characters in the string. The operand should be enclosed in single quote marks; if the string contains a single quote mark, code it four times.

No registers are affected.

Coding Examples:

```
DOSIN  'Here''''s a sample'  a string of length 15
```

DOSOUT

Define GS/OS Output String

Forms:

LAB DOSOUT LENGTH[,STRING]

Operands:

LAB - Label.

LENGTH - Maximum length of the string.

STRING - Default string.

Description:

DOSOUT defines a GS/OS output string. That is, a buffer length word, followed by a count word, followed by the ASCII characters which comprise the string. The buffer length is the length of the entire structure, including the four bytes of the buffer length and current string length; GS/OS uses this value to prevent trashing the memory that lies past the end of the string. The count word specifies the number of characters in the string. The string operand should be enclosed in single quote marks; if the string contains a single quote mark, code it four times.

The string operand is optional. If omitted, the macro creates a default string with a length of zero.

No registers are affected.

Coding Examples:

```
DOSOUT 255
DOSOUT 255,'file'
```

DSTR**Define ORCA String****Forms:**

```
LAB    DSTR    STRING,LENGTH
```

Operands:

LAB - Label.

STRING - Character string.

LENGTH - Maximum length of string.

Description:

An ORCA string is a set of characters which have both a maximum and current 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 than 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 STRING is the initial string which will be stored in the area. Both must be constants. STRING 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 STRING 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.

No registers are affected.

Coding Examples:

	DSTR	'They''''re here...'	initialize string constant
	DSTR	LENGTH=80	a string variable with
!			maximum length 80; needs
!			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 STRING

Operands:

LAB - Label.

STRING - String to define.

Description:

DW defines a "Pascal-type string." That is, a count byte followed by the ASCII characters which comprise the string. The count byte specifies the number of characters in the string. The operand should be enclosed in single quote marks; if the string contains a single quote mark, code it four times.

No registers are affected.

Coding Examples:

```
DW      'Here''''s a sample'  a string of length 15
```

INC2

Two-Byte Increment

Forms:

LAB INC2 N1

Operands:

LAB - Label.

N1 - Number to increment.

Description:

Increase the two-byte number N1 by one. The contents of the accumulator are lost if indirect addressing is used.

Coding Example:

```
INC2    NUM1            Increment 2-byte integer stored at NUM1
```

INC4

Four-Byte Increment

Forms:

LAB INC4 N1

Operands:

LAB - Label.

N1 - Number to increment.

Description:

Increase the four-byte number N1 by one. No registers are affected.

Coding Example:

```
INC4    NUM1            Increment 4-byte integer stored at NUM1
```

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
LAB	JVC	BP
LAB	JVS	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 65816 as an instruction, as well as those provided in this macro library.

A conditional jump can go to any memory location within the current bank.

No registers are affected.

Coding Example:

JNE	THERE	jump to THERE if condition code is "not
!		equal"

LA

Load Address

Forms:

LAB LA AD1,AD2

Operands:

LAB - Label.

AD1 - Place to load the address.

AD2 - Address to load.

Description:

The address specified by the second operand is loaded into the location specified by the first operand. Both must be absolute addresses. AD1 can be a multiple operand. The contents of the accumulator are lost.

Coding Examples:

	LA	NUM1,4	sets the two-byte integer NUM1 to 4
	LA	P1,THERE	loads the address of the label THERE
!			into the pointer P1
	LA	(P1,P2),THERE	loads both P1 and P2 with the
!			address THERE

LLA

Load Long Address

Forms:

LAB LLA AD1,AD2

Operands:

LAB - Label.

AD1 - Place to load the address.

AD2 - Address to load.

Description:

The address specified by the second operand is loaded into the location specified by the first operand. Both must be absolute addresses. AD1 can be a multiple operand. The contents of the accumulator are lost.

Coding Examples:

	LLA	NUM1,4	sets the 4-byte integer NUM1 to 4
	LLA	P1,THERE	loads the address of label THERE
!			into the pointer P1
	LLA	(P1,P2),THERE	loads both P1 and P2 with the
!			address THERE

LM

Load Memory

Forms:

LAB LM AD1,AD2

Operands:

LAB - Label.

AD1 - Place to load the byte.

AD2 - Byte to load.

Description:

The value specified by the second operand is loaded into the location specified by the first operand. AD1 must be an absolute address; AD2 can be absolute or immediate. AD1 can be a multiple operand. The contents of the accumulator are lost.

Coding Examples:

LM	NUM1,#4	sets the 1-byte integer NUM1 to 4
LM	(F1,F2,F3),#0	sets the 3 flags F1, F2, and F3 to 0
LM	F1,F2	sets F1 to the value of F2

LONG

Set Register State Long

Forms:

```
LAB    LONG    I
LAB    LONG    M
LAB    LONG    I,M
```

Operands:

LAB - Label
 I - index register (X and Y)
 M - accumulator (A)

Description:

Used to set the length of the specified register to sixteen bits. This macro also generates the appropriate assembler directive (LONGA ON or LONGI OFF) to match the state of the registers. No register contents are affected.

Coding Example:

```
                LONG    I                set X and Y index to 16-bits
```

LRET

Return from LSUB or CLSUB Macro

Forms:

```
LAB    LRET    [RET]
```

Operands:

LAB - Label.
 RET - Optional return value.

Description:

See the description of the SUB macro.

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)
```

LSUB

Local Subroutine With Parameters, Stack Frame

Forms:

LAB LSUB [PARMS] , SIZE

Operands:

LAB - Label.

PARMS - Parameter list.

SIZE - Size of direct page space.

Description:

See the description of the SUB macro.

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 value at ADR is shifted left NUM times. ADR can be the accumulator. NUM must be an integer constant. The size of the area shifted is either one or two bytes, depending on the size of the accumulator.

Coding Example:

MASL	A, 3	shift accumulator left three bits
MASL	MEM, 4	shift memory left four bits

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 value at ADR is shifted right NUM times. ADR can be the accumulator. NUM must be an integer constant. The size of the area shifted is either one or two bytes, depending on the size of the accumulator.

Coding Example:

MLSR	A, 3	shift accumulator left 3 bits
MLSR	MEM, 4	shift memory left 4 bits

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 beginning of the move range and proceeding toward the end. LEN can range from 0 to 65535; if 0, 65536 bytes are moved. LEN must be specified with absolute or immediate addressing. AD1 and AD2 can both use absolute addressing. AD1 can also use immediate addressing, in which case the entire destination range is set to the immediate value specified.

The data bank register is set to the value of the destination's bank. If the destination address is not in the data bank, precede the macro with a PHB directive to save the data bank, and follow it with a PLB directive to restore the data bank.

The contents of all registers are lost.

Coding Examples:

MOVE	#0,PAGE,#50	sets 50 bytes to 0, starting at PAGE
MOVE	HERE,THERE,Q	moves Q bytes from HERE to THERE

MOVE4

Move 4 Bytes of Memory

Forms:

LAB MOVE4 AD1,AD2

Operands:

LAB - Label.

AD1 - Source address.

AD2 - Destination address.

Description:

Four bytes are moved from AD1 to AD2. AD1 and AD2 can both use absolute or indirect addressing. AD1 can also use immediate addressing.

The MOVE4 macro is a special case of the MOVE macro that gives more addressing modes and more efficient code for this commonly used move length.

The contents of all registers are lost.

Coding Examples:

```
MOVE4    HERE,THERE
MOVE4    [4],{8}
```

PHx**Stack Push****Forms:**

LAB	PH2	ADR
LAB	PH4	ADR
LAB	PH8	ADR

Operands:

LAB - Label.

ADR - Address to be pushed. Can be immediate, indirect, or absolute.

Description:

These macros allow a two-, four- or eight-byte value to be pushed onto the hardware stack. Immediate or absolute addressing may be used. If absolute addressing is used, the accumulator is lost. The PH8 macro does not allow an expression to be used in the operand. This is because of a limitation in the DC directive.

The PH2 and PH4 macros support two special operand characteristics. If the operand starts with the character <, the macros assume the operand is a direct page operand, pushing the value using the PEA instruction instead of a LDA-PHA sequence. If the operand is the single character *, the macros do nothing; this capability is used by the tool macros when you specify an operand that is already on the stack.

Coding Examples:

PH2	#\$2A6	push a 2-byte constant value onto stack
PH4	VAR	push the 4-byte value at VAR onto stack
PH4	#VAR	push the address of VAR onto stack

PLx

Stack Pull

Forms:

LAB	PL2	ADR
LAB	PL4	ADR
LAB	PL8	ADR

Operands:

LAB - Label.

ADR - Address to place value. Must be absolute.

Description:

These macros allow a two-, four-, or eight-byte value to be pulled off the hardware stack. The accumulator contents are lost.

Coding Examples:

!	PL4	VAR	pulls 4 bytes off the stack and places them in VAR
!	PL2	\$02	pulls 2 bytes off the stack and places them in direct page location \$02

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. 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 125,000 machine cycles, or about 50,000 typical assembly language instructions. A clear symptom that enough time is not being allowed to elapse is unpredictable or clearly erroneous paddle readings.

The contents of all registers are lost.

Coding Examples:

```
PREAD #0,VAL1            read paddle 0 into VAL1
```

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)
```

RET

Return from SUB or CSUB Macro

Forms:

LAB RET [RET]

Operands:

LAB - Label.

RET - Optional return value.

Description:

See the description of the SUB macro.

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.

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 initialize 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 uncorrelated 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 II has an excellent source of random number seeds: the built-in clock.

The operand should be a two byte value; ideally, about 8 bits should be set and 8 should be clear. 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, or indirect addressing.

The contents of all registers are lost.

Coding Examples:

SEED	INT	uses the 2 bytes at INT as a seed
SEED	{P1}	uses the 2 bytes pointed to by P1
SEED	#8	uses 8 as a seed; good for debugging

SHORT

Set Register State Short

Forms:

```
LAB    SHORT I
LAB    SHORT M
LAB    SHORT I,M
```

Operands:

LAB - Label.
I - index registers (X and Y).
M - accumulator (A)

Description:

Set the length of the specified register to eight bits. This macro also generates the appropriate assembler directive (LONGA OFF or LONGI OFF) to match the state of the register.

The contents of the registers are not affected.

Coding Examples:

```
        SHORT I,M           sets X, Y, and A to eight bits
```

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, so that the link editor does not bring in that subroutine unless it is requested elsewhere. This macro finds its only use in subroutine libraries; examples of its use can be found in the subroutine library listings.

No registers are affected.

Coding Examples:

```
SOFTCALL   SYSEERROR
```

SUB **Subroutine With Parameters, Stack Frame**

Forms:

LAB SUB [PARMS], SIZE

Also Described Here:

LAB CSUB [PARMS], SIZE

LAB LSUB [PARMS], SIZE

LAB CLSUB [PARMS], SIZE

LAB RET [VALUE]

LAB LRET [VALUE]

Operands:

LAB - Label.

PARMS - Parameter list.

SIZE - Size of direct page space.

VALUE - Return value.

Description:

This family of macros is used to create assembly language subroutines that can be called from high-level languages, or that share properties of high-level language subroutines. The macros give you an easy way to implement three important features in your assembly language subroutines:

1. The macros handle parameters passed on the stack, including automatically removing the parameters from the stack when the subroutine returns.
2. The macros handle returning two byte values in the accumulator, and four byte values in the X-A register pair, just as high-level languages do. These values can be in local variables, global variables, or in the stack frame itself.
3. The macros create and dispose of local variable space, allocated from the stack, for a local direct page. This gives you a set of variables that are truly local to your subroutine. Since the variables (and parameters, for that matter) are accessed via direct page addressing, you can use this space for pointers, and do indirect addressing directly from the variables. Since you get true local variables, you can also easily implement recursive assembly language subroutines.

The macros are designed to be used in a pair, with a subroutine macro as the first line of executable code in your subroutine, and a return macro as the last executable line. The subroutine macros include SUB, CSUB, LSUB and CLSUB. The return macros include RET and LRET.

Starting with the most general case, you subroutine would begin with a SUB macro, and exit with a RET macro. The SUB macro has two parameters, a parameter list and the number of bytes of direct page work space you want to set aside. The parameter list is generally imbedded in parenthesis, which, in the ORCA macro language, automatically subscripts the first symbolic parameter. Each of the parameters consists of a length byte that specifies the length of the parameter in bytes, a colon, and the name of the parameter. Parameters can have any length from

1 to 9 bytes. If there are no subroutine parameters, leave the first macro parameter out entirely, but be sure and remember the comma as a place holder. The number of bytes of direct page space can be zero, but you must code some specific value. The first useable byte of direct page space will be byte 1. The total number of bytes of direct page space and memory used by the parameters must be less than 251 bytes.

Parameters are equated to their proper direct page location. You must not try to use direct page addressing inside your subroutine for direct page variables defined outside the subroutine, since the SUB macros change the value of the D register. (The RET macro restored the caller's D register before returning.)

The RET macro has an optional parameter. With no parameter, the RET macro just cleans up the stack frame created by the SUB macro, then returns to the caller. If you use a parameter, it should be a length byte, followed by a colon, followed by a parameter that is a valid operand for LDX and LDA instructions. The length must be either 2 or 4. If the length is 2, the value specified is returned in A. If the length is 4, the value is returned with the most significant word in the X register, and the least significant word in the A register.

Subroutines that use SUB and RET should be called using ORCA/Pascal's calling conventions; namely, call the subroutine with a JSL after pushing all parameters onto the stack in left to right order. The other versions of the macros provide variations on this basic theme. CSUB is used for ORCA/C's calling conventions; call the subroutine with a JSL, but push the parameters on the stack in right to left order. In both cases, the RET macro removes the parameters from the stack before returning to the caller.

LSUB is a variant on SUB that is used in assembly language programs that are smaller than 64K, or where all callers are in the same 64K bank as the subroutine. The LSUB macro looks and works just like SUB, but you call the subroutine with a JSR instruction instead of a JSL instruction. You must use LRET with LSUB; LRET returns with an RTS instead of an RTL. The CLSUB macro gives you C parameter passing order for these shorter macros.

Coding Examples:

```
*
* Recursive binary search
*
* Parameters:
*   aPtr - array pointer
*   target - value to search for
*   index - current index
*   size - current step size
*
Search    start
ptr1      equ    1                work pointer

          sub     (4:aPtr,4:target,2:index,2:size),4
          .
          .
          .
          ret     2:index
          end
```


Appendix A

Error Messages

This appendix lists the various unique error messages that are returned by the assembler or linker. Errors returned by the shell, editor, and utilities all originate with either GS/OS or one of the Apple IIGS tools; these errors are explained in the *Apple IIGS GS/OS Reference* and the *Apple IIGS Toolbox Reference*, respectively. Because of the context in which GS/OS and tool errors occur, it is generally easy to see what caused the error. If there is any doubt about the cause of an error, or if you do not have a copy of the Apple reference manuals, feel free to contact our customer service department for assistance.

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.

<u>Severity</u>	<u>Meaning</u>
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.
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. In the description following the error message, every possible cause for the error is explained, and ways to correct the problem are outlined.

Appendices

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 redefinitions 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
```

Appendix A: Error Messages

would produce this error, since P1 is set to A as a positional parameter, then to B as a keyword parameter.

Duplicate Segment [8]

Two segments with the same name were encountered during the same assembly. Either change the name of one of the segments, or remove the duplicate segment.

Expression Too Complex [8]

1. An expression contained a label whose value was defined with an EQU or GEQU that contained an expression. That expression also contained a label defined in the same way, and so on, for ten levels.

Reduce the nest level of expressions by coding some of the terms in long hand.

2. Too many parentheses were used.

The exact number of parentheses allowed depends on the type of expression you code, but is generally over twenty. If this error occurs, reduce the number of parentheses used in the expression.

3. A subscripted symbolic parameter was used to specify the index of another symbolic parameter, as in &A(&B(4)).

Subscripts for symbolic parameters cannot contain subscripted symbolic parameters. Form the subscript by first assigning the value to a different symbolic parameter, as in

```
&C      SETA  &B(4)
```

and then using &A(&C).

4. An ORG, OBJ, or DIRECT directive had an expression in its operand which did not resolve to a constant at assembly time.

Replace the expression with a constant.

5. GEQU appeared outside a segment with an operand that did not resolve to a constant.

Move the GEQU into a segment. Be sure the segment is assembled on all partial assemblies.

6. 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 in an expression, but generally the assembler will handle as many terms as will fit on a line. 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.

Appendices

Label Syntax [16]

1. 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.
2. 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 direct 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 direct page when the linker resolves the references, force direct page addressing by preceding the expression with a < character, as in

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 File Not In Use [2]

An MDROP was found that specified a macro file name that was never opened with an MLOAD or MCOPY, or that has already been closed with another MDROP.
Remove the extra MDROP.

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 embedded spaces.

Missing Label [2]

1. A DATA, ENTRY, EQU, GEQU, PRIVATE, PRIVDATA, or START was found that did not have a label. Since the purpose of these directives is to define a label, a label is required.
2. A directive that sets the value of a symbolic parameter was coded without a symbolic parameter in the label field. Since the purpose of these directives is to change the value of the symbolic parameter in the label field, a symbolic parameter is required.

Missing Operand [16]

The operation code was one that required an operand, but no operand was found.

Appendix A: Error Messages

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 one.

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:

AGO	AIF	ALIGN	APPEND	COPY
DIRECT	DYNCHK	EJECT	ERR	EXPAND
GEN	GEQU	IEEE	KEEP	KIND
LIST	LONGA	LONGI	MCOPY	MDROP
MERR	MLOAD	MSB	NUMSEX	ORG
PRINTER	RENAME	SETCOM	SYMBOL	TITLE
TRACE	65C02	65816		

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 ORGs 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 MEND [4]

A macro that did not have a MEND was expanded. The MEND directive is required at the end of a macro definition.

Appendices

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,

```
T      align    4
      start
      align    4
      end
```

is acceptable, but

```
T      align    4
      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

Appendix A: Error Messages

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]

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
```

Appendices

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:

ABSADDR	CASE	CODECHK	DATACHK	DYNCHK
ERR	EXPAND	GEN	IEEE	INSTIME
LIST	LONGA	LONGI	MSB	NUMSEX
OBJCASE	PRINTER	SYMBOL	TRACE	65C02
65816				

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.

Appendix A: Error Messages

AINPUT Table Damaged

This error will occur if a memory error has damaged the AINPUT table. Test your memory and try again.

File Could not be Opened

A GS/OS 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 path name 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.

GS/OS Errors

The assembler aborts if a GS/OS error is encountered on any file read or write. The GS/OS error message is reported.

Keep File Could Not be Opened

Either there was not enough memory to open the output file or a GS/OS 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 GS/OS restricts file names to fifteen characters.

Out Of Memory

The assembler ran out of memory. Add more memory, cut down on the size of the source files, cut down on the size of macro files, or reduce the number of symbols.

No END

The assembler aborts if any segment does not have a closing END directive.

Unable to Write to Object Module

A GS/OS 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 detects a nonfatal error, it prints:

1. the name of the segment that contained the error

Appendices

2. how far into the segment (in bytes) the error point lies
3. a text error message, with the error-level number in brackets immediately to the right of the message

Addressing error [16]

A label could not be placed at the same location on pass 2 as it was on pass 1.

This error is almost always accompanied by another error, which caused this one to occur; correcting the other error will correct this one. If there is no accompanying error, check for disk errors by doing a full assembly and link. If the error still occurs, report the problem as a bug.

Address is not in current bank [8]

The (most-significant-truncated) bytes of an expression did not evaluate to the value of the current location counter.

For short-address forms (6502-compatible), the truncated address bytes must match the current location counter. This restriction does not apply to long-form addresses (65816 native-mode addressing).

Address is not in zero page [8]

The most significant bytes of the evaluated expression were not zero, but were required to be zero by the particular statement in which the expression was used.

This error occurs only when the statement requires a direct-page address operand (range = 0 to 255).

Alignment and ORG conflict [8]

You have used both a fixed origin and an alignment factor, and the origin is not aligned to the boundary specified in the alignment factor. One should be changed or removed so the two values are compatible.

Alignment factor must be a power of two [8]

An alignment factor that was not a power of 2 was used in the source code. In ORCA Assembly language, the ALIGN directive is used to set an alignment factor.

Alignment factor must not exceed segment align factor [8]

An alignment factor specified inside the body of an object segment is greater than the alignment factor specified before the start of the segment. For example, if the segment is aligned to a page boundary (ALIGN = 256), you cannot align a portion of the segment to a larger boundary (such as ALIGN = 1024).

Appendix A: Error Messages

Code exceeds code bank size [4]

The load segment is larger than one memory bank (64K). You have to divide your program into smaller load segments.

Data area not found [2]

A USING directive was issued in a segment, and the linker could not find a DATA segment with the given name. Ensure that the proper libraries are included, or change the USING directive.

Duplicate label [8]

A label was defined twice in the program. Remove one of the definitions.

Duplicate segment [2]

Two segments were found in the same file with the same name. One of the names must be changed.

Expression operand is not in same segment [8]

An expression in the operand of an instruction or directive includes labels that are defined in two different relocatable segments. The linker cannot resolve the value of such an expression.

Illegal {AuxType} shell variable [4]

The linker found an {AuxType} shell variable, but it was not a valid decimal number or hexadecimal number.

Illegal {KeepType} shell variable [4]

The linker found a {KeepType} shell variable, but it was not a valid decimal number, hexadecimal number, or three-letter file type code.

Illegal shift operator [16]

An operation involving the shift operator could not be performed. This error results when an expression that is of a basically legal form uses the shift operator in a way that produces an unusable result. For example, shifting an address, performing some math operations, and then shifting the result would cause this error.

Invalid operation on relocatable expression [8]

The ORCA linker can resolve only certain expressions that contain labels that refer to relocatable segments. The following types of expressions *cannot* be used in an assembly-language operand involving one or more relocatable labels:

- A bit-by-bit NOT
- A bit-by-bit OR
- A bit-by-bit EOR

Appendices

A bit-by-bit AND
A logical NOT, OR, EOR, or AND
Any comparison (<, >, <>, <=, >=, ==)
Multiplication
Division
Integer remainder (MOD)

The following types of expressions involving a bit-shift operation *cannot* be used:

- The number of bytes by which to shift a value is a relocatable label
- A relocatable label is shifted more than once
- A relocatable label is shifted and then added to another value
- You cannot use addition where both values being added are relocatable (you *can* add a constant to a relocatable value).
- You cannot subtract a relocatable value from a constant (you *can* subtract a constant from a relocatable value).
- You cannot subtract one relocatable value from another defined in a different segment (you *can* subtract two relocatable values defined in the same segment).

Only JSL can reference dynamic segment [8]

You referenced a dynamic segment in an instruction other than a JSL. Only a JSL can be used to reference a dynamic segment.

ORG Location has been passed [16]

The linker encountered an ORG directive for a location it had already passed.

Move the segment to an earlier position in the program. This error applies only to absolute code, and should therefore be rarely encountered when writing for the Apple IIGS.

Relative address out of range [8]

The given destination address is too far from the current location.

Change the addressing mode or move the destination code closer.

Segment types conflict [2]

You have used a code, init, or direct page segment in a load segment that has already been flagged as a data segment. The linker is warning you that combining these data types can result in problems; for example, a data segment can be longer than 64K, but executable code cannot cross a bank boundary, so placing executable code in a data segment might produce a program that would crash.

Shift operator is not allowed on JSL to dynamic segment [8]

The operand to a JSL includes the label of a dynamic segment that is acted on by a bit-shift operator. You probably typed the wrong character, or used the wrong label by mistake.

TempORG not supported [4]

Appendix A: Error Messages

The TempORG field is not used by native Apple IIGS development systems. If you are importing object files from the MPW IIGS Cross Development environment, you must insure that all of the files have a TempORG of 0.

Unresolved reference [8]

The linker could not find a segment referenced by a label in the program.

If the label is listed in the global symbol table after the link, make sure the segment that references the label has issued a USING directive for the segment that contains the label. Otherwise, correct the problem by: (1) removing the label reference, (2) defining it as a global label, or (3) defining it in a data segment.

Terminal Linker Errors

Could not find library header in *filename*.

A file has a file type of LIB, but the linker could not find a library header. The file must be corrected before the linker can process it as a library.

Could not open file *filename*.

GS/OS could not open the file *filename*, which you specified in the command line.

Check the spelling of the file name you specified. Make sure the file is present on the disk and that the disk is not write-protected.

Could not overwrite existing file: *filename*.

The linker is only allowed to replace an existing output file if the file type of the output file is one of the executable types. It is not allowed to overwrite a TXT, SRC, or OBJ file, thus protecting the unaware user.

Expression too complex in *filename*.

An expression either used too many nested operations for the linker's stack or had some error in the form of the expression itself.

Check the original source file to see if the expression can be simplified. If there are no unwieldy expressions, look for errors that could corrupt the object file, like disk errors or disk accessories or Inits interfering with the system.

File read error: *filename*.

An I/O error occurred when the linker tried to read a file that was already open. This error should never occur. There may be a problem with the disk drive or with the file. You might have removed the disk before the link was complete.

File not Found *filename*.

The file *filename* could not be found.

Appendices

Check the spelling of the file name in both the KEEP directive and the LINK command. Make sure the .ROOT or .A file has the same prefix as the file specified in those commands.

File write error.

The linker encountered a GS/OS file output error trying to create or write the output file. The most common cause for this error would be a full disk or a write protected disk. If neither of these is the problem, check for a corrupted disk or bad blocks.

Illegal header value in *filename*.

The linker checks the segment headers in object files to make sure they make sense. This error means that the linker has found a problem with a segment header.

This error should not occur. Your file may have been corrupted, or the assembler or compiler may have made an error.

Invalid dictionary in *filename*.

The linker found a corrupted dictionary in a library file. The library file must be corrected before the linker can process it.

Linker version mismatch.

The object module format version of the object segment is more recent than the version of the linker you are using.

Check with the Byte Works to get the latest version of ORCA.

Must be an object file: *filename*.

Filename is not an object file or a library file.

Only one script file is allowed.

You cannot send two script files to the linker with a single compile command, as you can for, say, the assembler. Use one compile command for each link script.

Out of memory.

All free memory has been used; the memory needed by the linker is not available.

Script error: link aborted.

When the linker finds and reports an error in a linker script file, it does not continue processing; instead it generates this error and stops.

Stopped by open-apple .

You can stop the linker by holding down the ⌘ key and pressing the period. This actually forces a terminal error; this is the error message written by the linker.

Appendix A: Error Messages

Undefined opcode in *filename*.

An illegal operation code was found in the file. This error can be caused by any number of problems, including a buggy compiler or assembler, but the most frequent cause is a corrupted object file. Check for disk errors, memory errors, or programs (like desk accessories or Inits) that might be writing to memory that does not belong to them.

Appendix B

File Formats

Overview

The ORCA system makes use of four kinds of files: GS/OS TXT files, ORCA SRC files, ORCA OBJ files, and GS/OS EXE files.

Text Files

GS/OS 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 GS/OS 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. On the Apple IIGS, lines are limited to 255 characters. Both types of file can be created and changed by the system editor, described in Chapter 13. Most other GS/OS-based editors will also suffice, although it may be necessary to 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 GS/OS.

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 three 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-bit and sixteen-bit Apple II computers, and by early versions of ORCA/M and the Apple Programmer's Workshop on the Apple IIGS, has a one as the version number. The newest version, version two, is currently used by ORCA/M. A variant of this OMF format, known as version 2.1, adds an optional field to the end of the header. The optional field is not needed, nor is it created, by native development systems, although the ORCA utilities and linker will accept OMF 2.1 format files.

This version of ORCA/M generates and accepts version two of the object module format, and the linker and utilities, unless otherwise noted, will accept either version 1 version 2, or version 2.1 files.

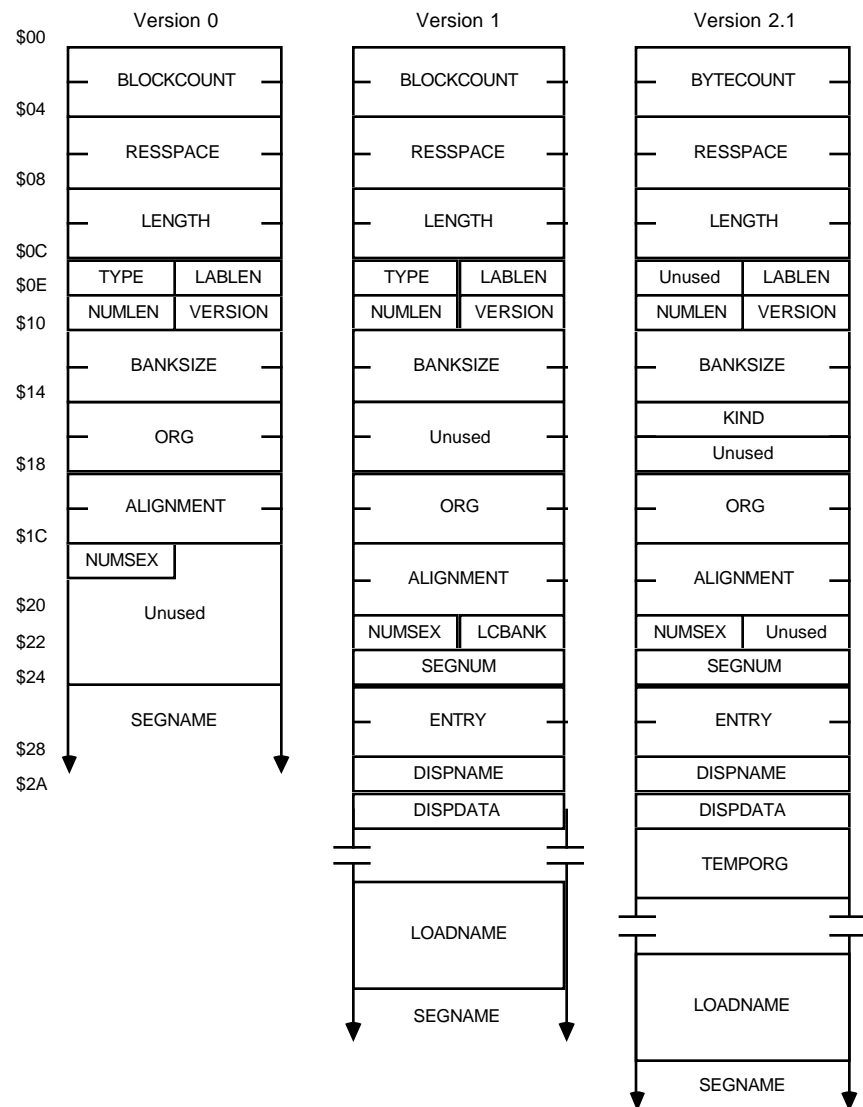
The description below describes version 2.1 object modules as used by this version of ORCA/M. Several features that deal with relocatable code and libraries have been omitted.

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. The object segment consists of two parts, the header and the body. The header describes the entire object segment. It contains such

Appendices

things as how many bytes the segment occupies, how many bytes it will add to the executable program image, and so on. The body of the object segment contains 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 three 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.



Appendix B: File Formats

<u>Name</u>	<u>Description</u>
BLOCKCOUNT	Number of blocks occupied by the segment. Versions 0 and 1 of the OMF aligned segments to 512 byte boundaries; this is the number of 512 byte blocks used by the segment.
BYTECOUNT	Number of bytes occupied by the segment. Version 2 of the OMF places one segment right after the end of the previous segment, to the length is a byte count, rather than a block count.
RESSPACE	Number of bytes to place at the end of the segment. These bytes will be filled with zeros.
LENGTH	Total number of bytes occupied by the segment in the executable file.
TYPE	In versions 0 and 1, this 1-byte field specifies the type and attributes of the segment.

<u>Bit</u>	<u>Meaning</u>
	Segment Type
0-4	\$00 code
	\$01 data
	\$02 jump table segment
	\$04 path name segment
	\$08 library dictionary segment
	\$10 initialization segment
	\$11 absolute-bank segment
	\$12 direct-page/stack segment
5-7	Segment Attribute
5	1=position independent
6	1=private
7	0=static; 1=dynamic

A segment can have only one type but any combination of attributes. For example, a position-independent dynamic data segment has TYPE=(\$A1).

KIND	In version 2, this 2-byte field specifies the type and attributes of the segment, replacing the TYPE field in the older OMF versions.
------	---

Appendices

<u>Bit</u>	<u>Meaning</u>
	Segment Type
0-4	\$00 code
	\$01 data
	\$02 jump table segment
	\$04 path name segment
	\$08 library dictionary segment
	\$10 initialization segment
	\$12 direct-page/stack segment
5-15	Segment Attribute
8	1=bank relative segment
9	1=skip segment
10	1=reload segment
11	1=absolute bank segment
12	1=cannot be loaded in special memory
13	1=position independent
14	1=private
15	0=static; 1=dynamic

A segment can have only one type but any combination of attributes. For example, a position-independent dynamic data segment has KIND=(\$A001).

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 of variable length.
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. This will be 0, 1 or 2.
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 ORG is zero, the segment is relocatable.
ALIGNMENT	Byte boundary to align the segment to. This value must be a power of two. If it is zero, the segment is not aligned.
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.

Appendix B: File Formats

LCBANK	Used on the Apple IIGS to indicate if a segment should be loaded into the language card area. This field is obsolete, and is no longer supported by Apple's loader.
SEGNUM	In the Apple IIGS load segment, this number is used by the loader to check the contents of the file. This field is used only in executable files. It is not created, used, or checked in OBJ files.
ENTRY	Displacement into the object segment where execution should begin.
DISPNAME	Displacement to the name field. Using a displacement allows the format to be expanded later. (See TEMPORG for an example.)
DISPDATA	Displacement to the data field. Using a displacement allows the format to be expanded later. It also allows for variable length segment names, without requiring the program reading the file to check the length of the segment name.
TEMPORG	Indicates a "temporary origin" for the MPW IIGS cross development assembler. This field is not used on the Apple IIGS. It is optional; if present, it must be set to 0.
LOADNAME	Name of the load segment. This field is always ten bytes long. It is the load segments for the current object segment. While the ten bytes are traditionally filled with printing ASCII characters, there is no real restriction imposed on the bytes.
SEGNAME	Name of the object segment.

The header is followed by the op code portion of the 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 NUMLEN=4, as E0 00 01 00 00.
------	---

Appendices

\$E1 **ORG** - The operand is a NUMLEN byte number indicating an absolute address to ORG to. If $\text{ORG} > *$ (* is the current location counter) zeros are inserted to reach ORG. If $\text{ORG} = *$, no action is taken. If $\text{ORG} < *$, the linker must backtrack in the binary load module. If $\text{ORG} < \text{START}$, where START is the first byte generated, an error results.

\$E2 **RELOC** - This is a relocation record, used in the relocation dictionary of a load segment. It is used to patch an address in a load segment with a reference to another address in the same load segment. It contains two 1-byte counts followed by two offsets. The first count is the number of bytes to be relocated, and the second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, then the number is shifted to the left, filling vacated bit positions with 0's. If the bit-shift operator is negative, then the number is shifted right.

The first offset gives the location (relative to the start of the segment) of the number that is to be relocated. The second offset is the location of the reference relative to the start of the segment; that is, it is the value that the number would have if the segment it's in started at address \$000000. For example, suppose the segment includes the following lines:

<u>location</u>	<u>line</u>
35	label anop
400	lda label+4

Label is a local reference to a location 53 (\$35) bytes after the start of the segment. When this segment is loaded into memory, the value of label+4 depends on the starting location of the segment, so the linker creates a RELOC record in the relocation dictionary for this value. Label+4 is two bytes long; that is, the number of bytes to be relocated is 2. No bit-shift operation is needed. The number to be calculated during relocation is 1025 (\$401) bytes after the start of the segment (immediately after the LDA, which is one byte). The value of label+4 would be \$39 if the segment started at address \$000000. The RELOC record for the number to be loaded into the A register by this statement would therefore look like this: (note that the values are stored low-byte first, as specified by NUMSEX):

E2020001 04000039 000000

which corresponds to the following values:

\$E2	operation code
\$02	number of bytes to be relocated
\$00	bit-shift operator
\$00000401	offset of value from start of segment
\$00000039	value if segment started at \$000000

Appendix B: File Formats

Certain types of arithmetic expressions are illegal in a relocatable segment. Specifically, any expression that cannot be evaluated (relative to the start of the segment) by the assembler cannot be used. The expression `LAB|4` can be evaluated, for example, since the RELOC record includes a bit-shift operator; however `LAB|4+4` cannot be used, because the assembler would have to know the absolute value of `LAB` in order to perform the bit-shift operation *before* adding 4 to it. Similarly, the value of `LAB*4` depends on the absolute value of `LAB`, and cannot be evaluated relative to the start of the segment, so multiplication is illegal in expressions in relocatable segments.

\$E3 INTERSEG - This record is used in the relocation dictionary of a load segment, and contains a patch to a long call to an external reference. The INTERSEG record is used to patch an address in a load segment with a reference to another address in a different load segment. It contains two 1-byte counts followed by an offset, a 2-byte file number, a 2-byte segment number, and a second offset. The first count is the number of bytes to be relocated, and the second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, then the number is shifted to the left, filling vacated bit positions with 0's. If the bit-shift operator is negative, then the number is shifted right.

The first offset is the location, relative to the start of the segment, of the number that is to be relocated. If the reference is to a static segment, then the file number, segment number, and second offset correspond to the subroutine referenced. The file number is always one.

For example, suppose the segment includes an instruction like

```
jsl    ext
```

where the label `ext` is an external reference to a location in a static segment. If this instruction is at relative address \$720 within its segment and `ext` is at relative address \$345 in segment \$000A in file \$0001, then the linker creates an INTERSEG record in the relocation dictionary that looks like this (note that the values are stored low-byte first, as specified by NUMSEX):

```
E3030020 07000001 000A0045 030000
```

which corresponds to the following values:

\$E3	operation code
\$03	number of bytes to be relocated
\$00	bit-shift operator
\$00000720	offset of instruction
\$0001	file number
\$000A	segment number
\$00000345	offset of subroutine referenced

When the loader processes the relocation dictionary, it uses the second offset to find the JSL, and patches in the address corresponding to the file number, segment number, and offset of the referenced subroutine.

Appendices

INTERSEG records are used for any long-address reference to a static segment

- \$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 current location counter. The name is followed by three fields.
The first field is the length attribute; it is a one-byte value in OMF versions 0 and 1, and a two-byte value in OMF version 2. If the value exceeds 255 in OMF version 0 or 1, or 65535 in OMF version 2, the value is set to 255 or 65535, respectively. This value indicates a field overflow.
The second value is a one-byte value. It is the type attribute for the label, and corresponds exactly to the type attribute used by the assembler.
The last field is a one-byte 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 fields, followed by an expression.
The first field is the length attribute; it is a one-byte value in OMF versions 0 and 1, and a two-byte value in OMF version 2. If the value exceeds 255 in OMF version 0 or 1, or 65535 in OMF version 2, the value is set to 255 or 65535, respectively. This value indicates a field overflow.
The second value is a one-byte value. It is the type attribute for the label, and corresponds exactly to the type attribute used by the assembler.
The last field is a one-byte 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. See EXPR (op code \$E8) for a description of an expression. (In version 0 of the OMF, this expression was always represented as a NUMLEN byte constant value.)
- \$E8** MEM - The operand is two absolute NUMLEN byte values specifying an absolute range of memory which must be reserved. This is not needed or supported on the Apple IIGS
- \$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:

<u>Operator</u>	<u>Description</u>
\$00	end of expression
\$01	signed integer add
\$02	signed integer subtract

Appendix B: File Formats

\$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 =
\$12	bit and
\$13	bit or
\$14	bit eor
\$15	bit not

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 LABLEN 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 are truncated to allow the final value to fit into the specified space must all be zero. If the truncated bytes are not zero, the linker will flag an error when the segment is processed. This is effectively a variable length implementation of a zero page protocol.

\$ED BKEXPR - same as \$EB, except that the bits are truncated to allow the final value to fit into the specified space must match the corresponding bits in the current 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

Appendices

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 indicating how many zero bytes to insert in the file at the current location counter.
- \$F2 LCONST - This record contains a 4-byte count followed by absolute code or data. The count indicates the number of bytes of data. LCONST is similar to CONST except that it allows for a much greater number of data bytes.
- \$F3 LEXPR - This record contains a 1-byte count followed by an expression. The expression is evaluated, and its value is truncated to the number of bytes specified in the count. The order of the truncation is from most significant to least significant. If the expression evaluates to a single label with a fixed, constant offset, and the label is in another segment, and that segment is a dynamic code segment, then the linker is allowed to create an entry for that label in the jump table segment. (The jump table segment provides a mechanism to allow dynamic loading of segments as they are needed.) Only a JSL instruction should generate an LEXPR record.
- \$F4 ENTRY - This record is used in a run-time library dictionary. It contains a two-byte segment number, followed by a label that is the name of the code segment or global entry point.
Run-time libraries are not used on the Apple IIGS, since tools serve the same purpose much more effectively.
- \$F5 cRELOC - This record is the compressed version of the RELOC record. It is identical to the RELOC record, except that the offsets are 2 bytes long rather than 4 bytes. The cRELOC record can be used only if both offsets are less than \$FFFF (65535). The following example compares a RELOC record and a cRELOC record for the same reference (for an explanation of each line of these records, see the discussion of the RELOC record):

RELOC cRELOC

\$E2	\$F5
\$02	\$02
\$00	\$00
\$00000401	\$0401
\$00000039	\$0039
(11 bytes)	(7 bytes)

Appendix B: File Formats

\$F6 **cINTERSEG** – This record is the compressed version of the **INTERSEG** record. It is identical to the **INTERSEG** record, except that the offsets are 2 bytes long rather than 4 bytes, the segment number is 1 byte rather than 2 bytes, and it does not include the 2-byte file number. The **cINTERSEG** record can be used only if both offsets are less than \$FFFF (65535), the segment number is less than 256, and the file number associated with the reference is 1. References to segments in run-time-library files must use **INTERSEG** records rather than **cINTERSEG** records.

The following example compares an **INTERSEG** record and a **cINTERSEG** record for the same reference (for an explanation of each line of these records, see the discussion of the **INTERSEG** record):

INTERSEG	cINTERSEG
\$E3	\$F6
\$03	\$03
\$00	\$00
\$00000720	\$0720
\$0001	
\$000A	\$0A
\$00000345	\$0345
(15 bytes)	(8 bytes)

\$F7 **SUPER** - Super records contain a series of **cRELOC**, **cINTERSEG** and **INTERSEG** records, compacted into a short, tabular form. The difference between a compacted OMF file and an uncompact OMF file is that compacted OMF files use **SUPER** records to reduce space and cut down on load time.

SUPER records are not covered in this appendix. For details on the format of **SUPER** records, see volume 2 of *Apple IIGS GS/OS Reference*.

Executable Files

While load files (executable files) and object files are used for entirely different purposes, their internal formats are the same. This makes it easier to write utility programs to work with the system. While the two file types share the same structure, there are limitations on which operation codes can appear in which file types, and some header fields are only used in one or the other.

The body of each load segment consists of two parts:

1. A memory image consisting of **LCONST** records and **DS** records containing all of the code and data that do not change with load address (with space reserved for location-dependent addresses). The **DS** records are inserted by the linker (in response to **DS** records in the object file) to reserve large blocks of space, rather than putting large blocks of zeros in the load file. These **DS** records are not used to save space in expressed files, which are formatted for load speed, rather than disk space.
2. A relocation dictionary that provides the information necessary to patch the memory image records at load time.

Appendices

When the segment is loaded into memory, each LCONST record or DS record is loaded in one piece, and then the relocation dictionary is processed. The relocation dictionary includes RELOC (or cRELOC) and INTERSEG (or cINTERSEG) records only: the RELOC records provide the information necessary to recalculate the values of location-dependent local references, and the INTERSEG records provide the information necessary to transfer control to another load segment. See the discussions of the RELOC and INTERSEG records in the last section for more information.

Appendix C

Custom Installations

This appendix is designed to help you install ORCA/M to take advantage of your specific hardware configuration. As shipped, ORCA/M is set up for people who have one or two 3.5 inch floppy disk drives. If that describes your system, you should make copies of the original disks, and use them just as they were shipped. If you have a hard disk, you can use Apple's Installer program to create an ORCA/M environment that suits your needs. Finally, this appendix describes the principal files that make up the ORCA development environment and the assembler; by studying this section, you can learn why we configured ORCA/M the way we did, and adjust the installation to suite your needs.

Installer Scripts

Apple's Installer can be used to install ORCA/M on your hard disk, either as a separate language or in combination with other ORCA languages. To run the installer, execute the Installer file from the ORCA.Extras disk. There are several installer scripts listed in the window that appears; these are described below. Select the one you want, select the disk that you want to install the program on from the right-hand list, and click on the Install button.

Please note that with the current version of Apple's Installer, you will have to select the installation script before you can pick a folder from the right-hand list.

New System

This is the basic, all-purpose installation script. It installs the full ORCA/M system and all of the macro files and help files that you don't have enough room for from a floppy-disk based system.

If you run a lot of software, you probably boot into the Finder or some other program launcher. In that case, you should probably install ORCA/M in a folder that is not at the root level of your hard disk.

If you plan to use your computer primarily for programming, you can set things up so you boot directly into ORCA/M. To do that, start by installing Apple's system disk without a Finder. (Apple's installer, shipped on their system disk and available free from your local Apple dealer, has an installation option to install the system with no Finder.) Next, install ORCA/M at the root level of your boot volume, making sure that ORCA.SYS16 is the first system file on the boot disk. System files are those files with a file type of S16 that end with the characters ".SYS16", as well as the files with a file type of SYS that end in the characters ".SYSTEM".

See also "ORCA Icons", "ORCA Pascal, C, Asm Libraries", ".PRINTER and .NULL", and "GSBug", below.

Appendices

ORCA Icons

If you use Apple's Finder as a program launcher, be sure and install the ORCA Icons. ORCA itself will show up as a whale, while the various source files, object files, and utilities will be displayed with distinctive icons.

.PRINTER and .NULL

The ORCA system disk comes with the .PRINTER and .NULL drivers installed, along with the relevant support files for the .PRINTER driver. If you will be running ORCA from your hard disk, and would like to install these drivers, select this option in the installer. It copies both drivers to your System:Drivers folder. It also copies the TextPrinterInit to your System:System.Setup folder, the TextPrinterCDev to your System:CDevs folder, and the TextPrinterCDA to your System:Desk.Accs folder.

GSBug

This option installs the Init version of the debugger in your system folder. Use it to quickly install GSBug on your hard disk.

ORCA Pascal, C, Asm Libraries

If you are using ORCA/M by itself, installing ORCA/M gives you all of the libraries you need.

If you are adding ORCA/C and ORCA/Pascal to your ORCA/M system, you must have a total of three library files in your library folder, and they must appear in the correct order. If you are missing any of the libraries, or if they are in the wrong order, you will get linker errors with either C, Pascal, or possibly with both languages. This installer script installs the libraries for C, Pascal, and assembly language in the correct order. (The libraries used by the assembler are also used by C and Pascal, so you get them anytime you use C or Pascal.) You can use this installer script before or after any of the other scripts.

You should not use this script unless you are installing C or Pascal. Installing extra libraries takes up a little more room on your disk; slows link times a little, since the linker has to scan an extra library; and uses up a little extra memory, since the library header is loaded by the linker. If you are using Pascal, but not C, you can go back later and delete the file ORCALIB from your libraries folder; if you are using C but not Pascal, you can delete the file PASLIB.

Update System

This script will update an old ORCA/M system or add ORCA/M to an existing ORCA/C or ORCA/Pascal system. All of the executable files from the ORCA/M disk are copied to your old system, but the LOGIN file, SYSCMND file, SYSEMAC file, SYSTABS file and SYSHELP file are not updated, since all of these may have been customized in your old system. Of course, if you are installing ORCA/M into an existing system that does not already have ORCA/M, you will need to add ASM65816 as a language to your SYSCMND file.

ORCA/M 2.0 is compatible with the compilers from ORCA/Pascal 1.3 and ORCA/C 1.2 and later. All of the other executable files from these languages are replaced when you update a system.

Appendix C: Custom Installations

Libraries for Pascal, C, and assembly language should all come from the same disk. The libraries included with this version of ORCA/M, for example, work fine with the 1.x versions of Pascal and C, updating the compilers so they use the new I/O conventions of GS/OS and the 2.0 version of the ORCA shell. The SYSLIB file from this version of ORCA/M cannot, however, be used with the PASLIB or ORCALIB files from earlier releases of the compilers.

See also "ORCA Icons", "ORCA Pascal, C, Asm Libraries", ".PRINTER and .NULL", and "GSBug", below.

Update System, No Editor

This installation script is basically the same as the "Update System" script, described above, but it doesn't install the text editor. It may seem silly at first to install a text system with no text editor, but there are a number of text-based editors available from third party sources; this installation option installs ORCA/M without removing your existing text editor.

See also "ORCA Icons", "ORCA Pascal, C, Asm Libraries", ".PRINTER and .NULL", and "GSBug", below.

RAM Disks

RAM disks come in a variety of sizes and flavors. One of the most common is a RAM disk allocated from the control panel of your computer. *We do not recommend using a RAM disk of this kind unless you have only one 3.5" floppy disk, and then we recommend keeping it small and using it only for temporary storage.* These RAM disks are allocated from the memory of your computer. ORCA/M can make very effective use of that memory if you let it – the system will perform better than if you try to copy parts of ORCA to your RAM disk. In addition, RAM disks allocated from main memory are easy to destroy from a program that is accidentally writing to memory that it has not allocated. While this is unusual in commercial programs, you may find that your own programs do this frequently during the development cycle. RAM disks that are not allocated from main memory, like Apple's "Slinky" RAM disk, are good for work space and even source code. The so-called ROM disks, or battery-backed RAM disks, should be treated as small hard disks. See the sections on installing ORCA/M on a hard disk for effective ways of using ROM disks.

Details About Configuration

In this section, we will explore why ORCA/M is configured the way it is by looking at what happens when you run ORCA/M, when ORCA looks for files, and where it looks for files. The material in this section is advanced information for experienced programmers. You do not need to understand this material for beginning and intermediate programming, and the entire section can safely be skipped.

You always start ORCA/M by running the ORCA.SYS16 file. This file contains the UNIX-like text based shell. The first thing the shell does after starting is to look for a folder called SHELL; this folder must be in the same location as the ORCA.SYS16 file. Inside this folder, the shell looks for an ASCII file (it can be stamped as a ProDOS TXT file or an ORCA SRC file) with the name SYSCMND; this is the command table. It is loaded one time, and never examined again. The shell must get at least this far, successfully loading the SYSCMND table, or it will stop with a system error.

Appendices

The next step taken by the shell is to set up the default prefixes. The shell is a GS/OS aware program, and expects the program launcher you use to set prefix 8 to the current prefix, and prefix 9 to the program's prefix. Prefix 8 is not changed if it has already been set by the program launcher, but the shell will set it to the same prefix as prefix 9 if prefix 8 is initially empty. The remaining prefixes default to prefix 9 plus some subdirectory, as show in the table below.

<u>prefix</u>	<u>set to</u>
13	9:libraries
14	9
15	9:shell
16	9:languages
17	9:utilities

The last step taken by the shell is to look in prefix 15 for a script file named LOGIN. To qualify, this file must have a file type of SRC, and a language stamp of EXEC. If the shell does not find a valid LOGIN file, it simply moves on; in other words, you can leave out the LOGIN file if you choose. Typically, this script file is used to set up custom aliases, set up shell variables, change the default prefixes listed above to other locations, and to execute PRIZM, the desktop development system. One thing this shows is that, as far as ORCA is concerned, the PRIZM desktop development system is actually nothing more than an application that you run from within the shell. Systems that default to the desktop programming environment do so by running PRIZM from within the LOGIN script, so PRIZM is executed as part of the boot process.

After executing the LOGIN script, the shell writes a # character to the screen and waits for further commands. If course, if PRIZM is executed from the LOGIN file, the shell never gets a chance to do this until you quit from PRIZM.

Prefixes 13 to 17 are initialized by the shell, but you can change them to point to other folders if you prefer. To understand how these prefixes are used, we'll look at the programs that currently use them.

When you use the EDIT command, the shell attempts to run a program named EDITOR; it expects to find an EXE file with this name in prefix 15 (the "shell" prefix). If the shell does not find an EXE file with the name EDITOR in prefix 15, it writes the message "ProDOS: File not found" and returns to the # prompt. The ORCA editor uses prefix 15 to locate the SYSTABS file (to set up the tab line), the SYSEMAC file (to set up the default editor macros), and the SYSHELP file (to write the editor help screen). The editor can function perfectly well without any of these files, although you will get a warning message each time you load a file if there is no SYSTABS file. When you cut, copy or paste text, the editor reads or writes a file called SYSTEMP to prefix 14; obviously, the editor will perform a lot faster on these operations if prefix 14 is set to point to a RAM disk.

A few other programs look at the SYSTABS file in prefix 15; PRIZM is another good example. No other use is currently made of prefix 15.

Prefix 14, which the editor uses as a work prefix, is also used by the shell when you pipe output from one program to become input to another program. The shell handles piping by creating a temporary file to hold the output of one program, reading this file as standard input for the next program. These pipe files are called SYSPIPE0, SYSPIPE1, and so forth, depending on how many pipes were used on a single command line.

When you use any of the commands to compile or link a program, the shell looks in prefix 16 for the compiler, assembler, or linker. For example, if you assemble an ASM65816 source file, the shall takes a look at the auxtype field for the file, which will have a value of 3. The shell then scans its internal copy of the SYSCMND file looking for a language with a number of 3, and

Appendix C: Custom Installations

finds one with a name of ASM65816. The shell then loads and executes the file 16:ASM65816; if it does not find such a file, it flags a language not available error .

Compilers and linkers make heavy use of prefix 13, which is not actually used by the shell. A convention has also gradually developed to put assembler macros and equate files in a folder called ainclude inside the library folder, although the assembler and MACGEN utility don't automatically scan this folder.

The linker also uses the library folder. When you link a program, especially one written in a high-level language, the program almost always needs a few subroutines from a standard library. The linker recognizes this automatically, and scans prefix 13 looking for library files. The linker ignores any folders or other non-library files it might find. When the linker finds a library file, it opens it, scans the files in the library to resolve any subroutines, closes the file, and moves on. The linker never goes back to rescan a library, which is why it is important for the libraries to be in the correct order.

Prefix 17 is the utility prefix. When you type a command from the shell, the shell checks to see if it is in the command table. If so, and if the file is a utility, the shell appends the name to 17: and executes the resulting file. For example, when you run the MAKELIB utility to create your own library, the shell actually executes the file 17:MAKELIB, giving a file not found error if there is no such file. Utilities are not limited to EXE type files; you can make an SYS file, S16 file, or script file a utility, too.

Prefix 17 is also used by the help command. When you type HELP with no parameters, the help command dumps the command names from the SYSCMND table. When you type HELP with some parameter, like

```
help catalog
```

the help command looks for a text (TXT) or source (SRC) file named 17:HELP:CATALOG, typing the file if it is found. In other words, you can use the help command to type any standard file, as long as you put that file in the HELP folder inside of the utilities folder.

All of the files that were not mentioned in this section can be placed absolutely anywhere you want to put them – since none of the ORCA software looks for the files in a specific location, you have to tell the system where they are anyway. It might as well be a location you can remember, so pick one that makes sense to you.

All of this information can be put to use for a variety of purposes. For example, by installing the Finder, BASIC.SYSTEM, and any other programs you use regularly as utilities under ORCA, you can boot directly into ORCA's text environment (which takes less time than booting into the Finder) and use ORCA as a program launcher. You can also split the ORCA system across several 3.5" floppy disks by moving, say, the libraries folder to the second disk, setting prefix 13 to point to the new disk from within your LOGIN file.

Appendix D Licensing

Using the .PRINTER Driver

Programs that print by opening .PRINTER and writing through GS:OS will not work unless the .PRINTER driver is installed in the SYSTEM:DRIVERS folder of the boot disk. If you wish to distribute a program that requires this driver, you must include an appropriate copyright statement with your program. This copyright statement should appear on the distribution disk, and should read:

.PRINTER Driver Copyright 1991, Byte Works Inc.

You may only distribute the printer driver with other software, and the software must be created in whole or in part with an ORCA language. There is no fee for using the .PRINTER driver.

Macros

Macros may be used in your own programs without restriction, so long as the source code for the macros are not distributed with the program.

If you would like to distribute the macros with your program, the following copyright message must appear in the macro source file:

Copyright 1991, Byte Works, Inc.

Using SYSLIB

If your program must be linked with the ORCA/M SYSLIB library file, you are using copyrighted libraries in your program. In that case, the following copyright message must appear in the same location as any other copyright information in your program:

Contains libraries from ORCA/M, Copyright 1991, Byte Works, Inc.

There is no fee for using the libraries in your program.

Appendix E

Differences Between ORCA/M 2.0 and ORCA/M 1.0

This appendix summarizes the major differences between using ORCA/M 2.0 under GS/OS and using ORCA/M 1.1 under ProDOS 16. In many cases, this appendix only points out that a difference exists and gives a broad overview of the difference; you can refer to the main body of this reference manual for further details.

GS/OS

Path Names

GS/OS now supports longer file names, longer path names, file names with new characters, and the use of lowercase characters in file names. All of the programs in ORCA/M fully support these changes. Specifically:

- ORCA allows path names up to 32K in length.
- The `:` separator can be used instead of the `/` separator. The `/` separator can also be used, although you cannot mix the two separators in the same path name. For example, these two path names are equivalent, and either is acceptable:

```
/Disk/Folder/File  
:Disk:Folder:File
```

- File names can be up to 32K in length, and can contain any typeable character except `':'`. If the file name contains spaces, the `'?'` character, or the `'='` character, you must inclose the file name in quotes. While ORCA and GS/OS support a wide variety of file names, not all FSTs support all characters; in particular, the ProDOS FST, which is currently used to format floppy disks and hard disks, only allows 15 characters in a file name, forces the characters to be alphanumeric or the period, and requires that the first character be alphabetic.
- Lowercase letters are accepted throughout the ORCA environment. When a file is created using lowercase letters, the file name on disk will use the same case. Commands that print file names, like CATALOG, use the same case as is found on disk. The RENAME command can be used to change the case of existing file names. While uppercase and lowercase letters are allowed for readability, case is not significant, so File, FILE and file all refer to the same file.

Because of these changes, the ORCA Shell is set to grow with GS/OS. As this manual is written, the only FSTs that are available are for ProDOS, AppleShare and CD ROM drives, but the shell will support future FSTs automatically as GS/OS grows.

Appendices

Standard I/O

The shell now uses the GS/OS .CONSOLE driver for input and output. Due to current limitations in GS/OS, your programs must still use the text tools for input and output if you want your input and output to be redirectable under the shell. We hope this will change in the relatively near future.

Numbered Prefixes

ProDOS supported eight numbered prefixes, from 0 to 7. Each of these prefixes was limited to 64 characters, and all but prefix 7 had preassigned uses, either for ProDOS itself or for the ORCA system.

GS/OS still supports prefixes 0 to 7 for backward compatibility, but these prefixes have all been superseded by longer prefixes, numbered 8 to 31. GS/OS does not support use of both the prefixes from 0 to 7 and the prefixes from 8 to 31 in the same application.

The following table shows the old prefix numbers, as well as the new prefix number, and the standard use for the prefix.

<u>Old Prefix</u>	<u>New Prefix</u>	<u>Use</u>
0	8	Default prefix.
1	9	Program prefix; set to programs location when executed.
2	13	Library prefix.
3	14	Work prefix.
4	15	Shell prefix (formerly called the ORCA system prefix).
5	16	Languages prefix.
6	17	Utilities prefix.
7	18	No prespecified use.

When you are using the ORCA shell, prefixes 0 to 7 will be set to null. If you attempt to use one of these prefixes, the shell automatically maps the prefix into the new GS/OS prefix, leaving the old ProDOS prefix set to null.

When a program is executed by the shell (or any other GS/OS program launcher), it examines the auxiliary file type to determine if the program is GS/OS aware. If so, the prefixes are left exactly as they were in the shell. If not, the new prefixes are mapped back into the old ProDOS prefixes before the program is executed, then mapped back when the program returns control to the shell. If any of the new prefixes are too long to map into the shorter ProDOS prefixes, the shell flags an error and refuses to execute the program.

Shell

Command Line Length

The shell no longer limits the expanded length of a command line to 255 characters. Instead, a command line can expand to 64K characters. File name expansion, which severely limited the number of files that could be linked with the old limit of 255 characters on a line, can also utilize the full 64K characters, allowing you to link virtually any number of files.

In script files, it is possible to use the \ character to concatenate lines, building up a long line from a series of shorter ones in the editor.

Appendix E: Differences Between ORCA/M 2.0 and ORCA/M 1.0

The command line editor is still limited to 255 characters, but shell variables can be used to build up a longer line.

Shell Variable Length

Shell variable values can be up to 64K characters, as opposed to the old limit of 255 characters.

New .PRINTER Driver

The built-in .PRINTER driver from the old shell has been removed, and replaced with a new GS/OS driver, also called .PRINTER. From the shell, this means that you use the printer in I/O redirection exactly as you always did, but the new driver means you can also use the text printer from outside the shell.

The old shell variables used to configure the shell's built-in printer driver are no longer used.

New .NULL Driver

The shell has another new driver, called .NULL. This driver accepts characters and does nothing with them. It's a lot more useful than it sounds! This driver is generally used in scripts, where you can redirect unwanted output to .NULL so it won't show up on the console.

Shell Prefix

The old ORCA system prefix (prefix number 4, now 15) has been renamed to shell to avoid confusion with the GS/OS system prefix.

Larger Default Stack

The default stack size for a program launched by ProDOS or the ORCA 1.1 shell was 1K. The default stack size for a program launched by GS/OS or the ORCA 2.0 shell is 4K.

New Shell Variables

The shell supports several new shell variables to control various defaults.

<u>Shell Variable</u>	<u>Use</u>
Prompt	Replace the shell's # prompt with your custom prompt.
Insert	The shell's line editor defaults to over strike mode; this variable can be used to force the shell to start in over strike mode.
Separator	Used to change the default output separator from : to / (or some other value).

New Command Line Editor

The command line editor uses the standard Apple cursors for insert and over strike mode, and supports several new features, including word tab right, word tab left, forward delete, and undo.

Appendices

New or Changed Commands

The table below summarizes additions and changes to the shell commands. New commands are flagged with an asterisk.

<u>Command</u>	<u>Changes</u>
CATALOG	Supports the invisible bit. Shows extended files. Many new flags control output and provide more information. Supports any valid GS/OS FST.
CHANGE	-P flag can be used to suppress progress information.
COPY	Copies files with resource forks. Can copy only the resource fork, or only the data fork.
CRUNCH	Supports OMF 2.0. A new flag lets you suppress progress information.
DELETE	Can delete a directory and its contents. -P flag can be used to suppress progress information. -W flag can be used to suppress file not found warnings.
DEVICES*	Lists GS/OS devices. A variety of flags control the format and extend of the output.
DISABLE	Supports the invisible bit. -P flag can be used to suppress progress information.
DUMPOBJ	Supports OMF 2.0.
ECHO	A new flag allows you to suppress a carriage return. A new flag allows you to suppress expansion of tabs.
EDIT	Accepts multiple input names.
ENABLE	Supports the invisible bit. -P flag can be used to suppress progress information.
ENTAB*	Replaces runs of spaces with tab characters.
ERASE*	Erase a disk.
EXISTS*	See if a file exists.
FILETYPE	Allows you to change the auxiliary file type. -P flag can be used to suppress progress information.
HELP	Supports tabs in help files.
HOME*	Clear the screen and home the cursor.
INIT	Supports multiple FSTs.
INPUT*	Read a response from the user.
LINKER*	New linker script language.
MACGEN	Supports tabs. A new flag lets you suppress progress information.
MAKEBIN	Supports OMF 2.0. A new flag lets you suppress progress information.
MAKELIB	Supports OMF 2.0. A new flag lets you suppress progress information.
MOVE	Moves extended files.
NEWER*	Checks to see if a file is newer than a list of files.
PREFIX	-C flag allows you to set a prefix that is not online.
SET	Allows shell variable values up to 64K in length.

Appendix E: Differences Between ORCA/M 2.0 and ORCA/M 1.0

SHOW	Lists the boot prefix (*) and the network area prefix (@). Supports prefixes up to number 31. Times are now formatted in accordance with control panel settings.
SHUTDOWN*	Ejects disks and shuts down the computer.
SWITCH	-P flag can be used to suppress progress information.
TOUCH*	Set modification date to current date.
TYPE	Expands tabs for correctly formatted screen output. Supports SYSTABS file for tab expansion. Tab expansion can be suppressed.

Editor

The editor supports files up to the length of available memory.

Lines are no longer limited to 255 characters. When a line exceeds 255 characters, the line is wrapped using a soft carriage return. This works just like most text processors.

Tabs are now supported.

The editor can edit up to ten files at one time.

A new flag in the SYSTABS file lets you enter the editor in insert mode by default.

All time-consuming commands can now be canceled with the ⌘. command.

Mouse support has been added (although you can still use all commands from the keyboard).

Search and Search/Replace have been updated. They now support searching for whole words, special whitespace modes that make searching for strings with tabs easier, and case sensitive searches. Internal enhancements have increased the speed of these commands, too.

You can display hidden characters, like tabs and end of line marks.

The standard Apple cursors for insert and over strike modes are now used.

A new command allows blocks of text to be shifted right and left.

ASM65816

The KIND directive now supports the 16 bit segment kinds of OMF 2.0.

Object files use OMF 2.0, reducing the amount of disk space required for OBJ files.

The assembler supports tabs, using the appropriate tab line from the SYSTABS file.

The -P flag allows you to suppress the progress information normally printed by the assembler.

LINKER

The linker supports a limited scripting mode.

The -P flag can be used to suppress progress information.

The {AuxType} shell variable can be used to set the auxiliary file type of the executable file.

Rez Compiler

Apple's Rez resource compiler is now a standard part of the ORCA/M package.

Appendices

GSBug

Apple's GSBug debugger now ships as a standard part of ORCA/M.

New Utilities

Several new utilities have been imported from Apple's "Tools and Interfaces for APW," including COMPACT, DEREZ, DISKCHECK, EXPRESS and RESEQUAL.

Macros

A complete new set of tool macros are available for the tools. These macros let you type tool parameters on the same line as the macro, rather than pushing parameters before the macro call. For example, drawing a box is a mere 5 lines:

```
MoveTo #10,#10
LineTo #100,#10
LineTo #100,#100
LineTo #10,#100
LineTo #10,#10
```

There are also several new macros, including: a set of macros for high-level language style parameter passing and recursion (SUB, CSUB, LSUB, CLSUB, RET and LRET), a more efficient move macro for four byte values (MOVE4), improved stack push macros (PH2 and PH4), and macros for creating GS/OS strings (DOSIN and DOSOUT).

Special Characters

- & character 26
- &SYSCNT symbolic parameter 67, **341**
- &SYSDATE symbolic parameter **342**
- &SYSNAME symbolic parameter **342**
- &SYSOPR symbolic parameter **342**
- &SYSTIME symbolic parameter **342**
- {0} shell variable 78, 81, 82, 83, **104**
- {1}, {2}, ... shell variables 81, 82, 83, **104**
- {AuxType} shell variable **104**, 142, 187
- {CaseSensitive} shell variable **104**
- {Command} shell variable 81, 82, 83, **104**
- {Echo} shell variable **104**
- {Exit} shell variable 81-83, 86, **104**, 149, 153, 395
- {Insert} shell variable **104**
- {KeepName} shell variable **105**, 118, 184, 187, 302, 411, 435, 483
- {KeepType} shell variable **105**, 142, 187
- {Libraries} shell variable **105**, 184
- {LinkName} shell variable **105**, 148, 187
- {Parameters} shell variable 81, 82, 83, **105**
- {Prompt} shell variable **105**
- {Separator} shell variable **106**
- {Status} shell variable 81, 82, 83, 86, **106**, 141, 149, 153

Numbers

- 6502
 - cpu xxv, 50, 309, 321, 334
 - writing code for 325, 334
 - writing programs for 50, 150
- 65802
 - cpu xxv
- 65816
 - cpu xxiv, xxv, 48, 50, 190, 309, 321, 329, 344
 - directive 51, 304, **334**, 344, 475, 478
 - instruction set 23, 56, 308, 332, 334, 337, 355, 442, 478
- 658881 card 324
- 65C02
 - cpu xxv, 50, 321, 334
 - directive 51, 304, **334**, 344, 475, 478
 - instruction set 478
 - writing code for 325, 334
 - writing programs for 50, 150

A

- ABORT command 9, 41, 117, 304, 438
- ABSADDR directive 303, **318**, 344, 478
- absolute addressing 309, 310, 325, 331, 353, 359
- ABSx macro **363**
- ACTR directive **346**, 472
- ADD4 macro 53, 57
- Address not in current bank error 190
- addresses **322**
- addressing modes 50, 309, 359
 - assembler 473, 481
- ADDx macro **364**
- AGO directive 65, **347**, 472, 475, 477, 482
- AIF directive 65, 66, 343, 345, 346, **348**, 352, 472, 475, 477, 482
- AINPUT directive 107, **348**, 477, 479
- ALIAS command **115**, 502
- aliasing 90, 112, 120, 137, 140, 160
- ALIGN directive **318**, 345, 475, 476, 480
- alphabetizing directories 125
- ALTCH macro **373**, 380
- AMID directive 64, **349**, 477
- ANOP directive 23, 28, **319**, 357, 406, 413
- APPEND directive 45, 116, 150, 299, **319**, 337, 347, 351, 475
- appending to the end of a file 40, **107**, 425
- Apple Computer, Inc. xxiv
- Apple IIGS computer xxv
- AppleShare 146
- AppleSoft 150, 323, 328
- APW 70, 73, 487
- APW C 124
- APW macros 56, 357
- ASCII 30, 47, 102, 104, 134, 154, 318, 323, 330, 355, 360, 373-375, 377, 380, 384, 398, 421, 423, 427, 439, 440, 447-450, 487, 490
- ASEARCH directive 64, **349**, 477
- ASL2 macro **441**
- ASM6502 language 96
- ASM65816
 - command 1
 - language 9, 81, 96
- ASM65816 command **116**
- ASML command 41, 42, 87, 101, 105, **116**, 119, 148, 156, 184, 185, 299, 300, 305, 306, 412, 433

Appendices

ASMLG command 41, 43, 105, 116, 148, 156, 184, 299, 300, 330
ASSEMBLE command 11, 41, 86, 116, 152, 191, 299, 300, 412
assembler 112
assembler directives 23, 45-51, 303, **317-334**
assembler listing 41, 46, 117, 301, 304, **305**, 326, 329, 334
assembler variables 26, 28
assembling a program 3, 40, 90, 116, **119**, 123, **125**, **299-306**
assembly language
 macros 24
 statements 23
attributes 65-67
AUX field 487
auxiliary file type 104, 122, 142

B

back up xxiii
bank boundary 48, 190, 481
bank relative programs 147, 186
BELL macro **374**
BGE instruction 308
BGT macro **442**
BIN files 48, 150
binary constant 313, 476
binary data **323**
bit shifting 312, 457, 458, 481, 482, 492, 493
BLE macro **442**
BLOAD command 150
block boundary 488
blocks 121, 122
BLT instruction 308
BMI macro 443
boot prefix 38, 97, 98
booting ORCA xxiii
branching
 assembler 65, 66, 365, 442, 445, 452, 476
 conditional assembly 472
 conditional assembly language 477, 482
 EXEC files 86, 103
 object module format 496
BREAK command 86, **119**, 149
BRUN command 150

BUTTON macro **443**
byte order 29, 330, 490, 493

C

C 97, 469, 501
CASE directive 47, 303, **319**, 344, 478
case sensitivity 174
 assembler 24, 47, 319, 330
 linker 330
 shell 35, 86, 103, 104, 106
CAT command (See CATALOG command)
CATALOG command 8, 10, 11, 96, 107, 112, **120**, 125, 132, 142, 158
CDA 75
CDevs 75
CHANGE command 9, 81, **123**, 138
ChangeVector shell call **388**
character constant 315
character data **323**
Classic Desk Accessory 75
CLEOL macro **374**
CLEOS macro **375**
CLSUB macro 443, **469**
CMPL command 41, 101, 116, **123**, 148, 184, 299, 300
CMPLG command 41, 101, 116, **123**, 148, 184, 299, 300, 411
CMPx macro **365**
CNVxy macros **444**
code segment 26, 30, 31, 42, 45, 59, 67, 118, 152, 189, 190, 300, 304, 328, 332, 333, 342, 473, 475, 487
CODECHK directive 50, 303, **319**, 344, 478
command line 40, 70, 77, 96, 102-105, 107, 140, 156, 183, 306, 391, 483
 length 508
 prompt 105
command list 10, 144
command numbers 406
command table 7, 39, 72, 89, **112**, 124, 144, 405
commands (See shell commands)
COMMANDS command 72, 94, 112, **124**
COMMENT command 10, 107, **160**
comments 26
 assembler 46

Appendix E: Differences Between ORCA/M 2.0 and ORCA/M 1.0

- assembly language 25, 307, 315, 333, 474
- EXEC files 107
- COMPACT command **124**
- compaction 147, 186
- COMPILE command 41, 101, 116, 125, 299, 300, 411
- compiler
 - installing new 72
 - writing new 73, 400
- COMPRESS command **125**, 158
- conditional assembly 26, 65, 353
 - branching 347, 348
 - looping 346
- console driver 390
- ConsoleOut shell call 388, **390**
- constants 27, 45, 309, 312, 320, 325-327, 449, 474, 478, 482, 494
- CONTINUE command **125**
- control panel devices 75
- COPY command 11, 79, 85, 86, 101, **126**
- COPY directive 45, 150, 299, 303, **319**, 337, 347, 351, 475
- COUNT attribute 65, **343**, 478, 495
- COUL macro **376**
- CPA instruction 308
- CPM 146
- CPU cycles 328
- CREATE command **128**
- creation time 122
- cross assembler 332
- CRUNCH utility 86, 96, 99, **128**
- CSUB macro 445, **469**
- current prefix 98, 120, 125
- cursor positioning 379, 380, 390

D

- data 28, 320, 325
 - character 29, 30
 - hexadecimal 29
 - integer 29
- DATA directive 23, 25, 31, 48, 118, 190, 303, 304, 318, **320**, 328, 331, 332, 474, 481
- data fork 126, 226
- data segment 31, 48, 320, 334, 473, 481, 483, 487, 494, 496

- DATACHK directive 50, 303, **320**, 344, 478
- date 122
- DBcn macros **445**
- DC directive 23, 28, 30, 46, 47, 66, **320**, 326-328, 330, 342, 344, 345, 357, 413, 422, 426, 428, 430, 431, 436, 437, 439, 468
- debugger (see GSBUG)
- DEC2 macro **446**
- DEC4 macro **446**
- decimal constant 314
- decimal data 476
- default prefix 508
- DELETE command 101, 125, **128**
- DeRez 195-197, 200, 201, 204, 207, 214, 216, 220, 226
- DEREZ command **129**
- desktop programming 388
- desktop programs 69
- DETAB 160
- device names 37, **100**, 126, 157, 396, 414
- device numbers 100, 407, 429
- devices 37, 40, 131, 157, 360, 373, 374, 375, 376, 377, 380, 382, 384, 407
 - .CONSOLE 40, 100, 107, 407, 426, 429
 - .NULL 111
 - .PRINTER 40, 100, 107, 108, 306, 426
- DEVICES command **131**
- DIRECT directive 303, **325**, 473, 475
- direct page 28, 31, 305, 309, 310, 325, 327, 461, 472, 474, 480, 495, 509
- Direction shell call **391**
- directives (See assembler directives)
- directory walking 36, **100**, 396, 414
- DISABLE command 11, **131**, 138
- disassembler 121
- disassembly 133, 137
- disk caching 400
- disk copying 126
- disk size 121
- DISKCHECK command **132**
- displaying files (See the TYPE command)
- DIV4 macro 53
- DIVx macro **366**
- DOS 125, 158
- DOS (Apple) 146

Appendices

DOSIN macro **447**
DOSOUT macro **448**
drivers 330
DS directive 23, 30, 55, **325**, 331, 345,
406, 413, 422, 426, 436, 438, 439, 440
DSTR macro 377, 382, **449**
DUMPBOJ utility **137**
DUMPOBJ command 101
DUMPOBJ utility 115, **133**
duplicate label 67, 320
DW macro 66, 344, **450**
dynamic segments 325, 482, 489, 490, 496
DYNCHK directive **325**, 344, 475, 478

E

ECHO command 80, 119, **137**, 144
EDIT command 2, 11, 13, 81, 98, **138**,
415, 502
editor 2, 13-21, 90, 112, 161-181, 301, 304,
408, 412, 433, 435, 478, 487
about command **167**
arrow keys 2, 14, 164
auto-indent mode **162**, 172, 180
beep the speaker command **167**
beginning of line command **168**
bottom of page command 15
bottom of screen command 162, **168**
buttons 166
check boxes 166
close command **168**
continue searching macro 17
control underscore key **162**
copy command 20, **168**
create macros command 164
cursor down command 162, **168**
cursor left command 163, **168**
cursor movement 2, 14-16
cursor right command 163, **168**
cursor up command 162, **168**
customizing **179**
cut command 20, **169**
define macros command **169**
delete character command **169**, 179
delete character left command **169**, 179
delete command 21, **169**
delete current character command 16
DELETE key 2, 16
delete line command 16, **169**, 179

delete to end of line command 16, **169**,
179
delete word command **170**, 179
deleting characters in macros 164
dialogs 165
editline controls 165
editline items 165
end macro definition command 164
end of line command 15, **170**
ESCAPE key 20, 163
escape mode 15, **162**
executing macros **165**
exit macro creation command 165
help command 13, **170**
hidden characters 163
insert blank lines command 17, 162,
170
insert mode 17, **161**
insert one space command 17, **170**
installing new 72
line length **161**
list controls 166
macro keystrokes **164**
macros **164**
modes 179
mouse 167
moving through a file 15, **173**
multiple files 171, 176, 177
new command **170**
open Apple key 162
open command **171**
over strike mode **161**, 180
paste command 20, 168, **172**
quit command 3, 21, **172**
remove blanks command 17, **172**
repeat counts 15, 16, 17, **162**, **172**
RETURN key 2, 14, 20, 163, **172**
save as command **173**
save command 3, **173**
screen move commands 162
scroll down one line command **174**
scroll down one page command 15, **174**
scroll up one line **174**
scroll up one page command 15, **174**
scrolling 15
search and replace down command 19,
175
search and replace up command 19, **176**
search down command 17, **174**

Appendix E: Differences Between ORCA/M 2.0 and ORCA/M 1.0

- search up command **175**
- select file command **176**
- select mode 180
 - by character **162**, 163, 172
 - by line 20, **162**
- set/clear auto-indent mode command **178**
- set/clear escape mode command **178**
- set/clear insert mode command **178**
- set/clear select mode command **178**
- set/clear tab stops command **177**
- setting defaults **179**
- shift left command **177**
- shift right command **177**
- start of line command 15, 162
- status banner 2, 14, 117
- switch files command **177**
- tab command 163, **178**
- TAB key 2, 14
- tab left command 163, **178**
- tab mode 180
- tabs 163, 166
- top of page command 15
- top of screen command 162, **179**
- undo command 169
- undo delete buffer 169, 179
- undo delete command **179**
- version **167**
- word left command 163, **179**
- word right command 163, **179**
- EDITOR file 99
- EJECT directive 46, 47, **326**, 475
- ELSE clause of IF command 86, 87, **138**, 145
- ELSE IF clause of IF command 86, 145, 149
- emulation mode 136
- ENABLE command 11, 132, **138**, 142
- END command 84, 85, 86, 87, 119, 125, **139**, 144, 145, 149
- END directive 23, 26, 31, 62, 117, 301, 304, **326**, 336, 479
- ENTAB utility **139**
- ENTRY directive 23, 31, **326**, 332, 474
- EOF 122
- EQU directive 23, 27-28, 45, 309, 312, **326**, 345, 472, 473, 474
- ERASE command **139**
- ERR directive 46, 47, 303, **327**, 344, 475, 478
- error levels 410, 433
 - assembler 50, 330, 353, 471
 - linker 479
- error output 40, 69, 97, 108, 373-376, 379-382, 384, 425, 430, 508
- error reporting 410
 - assembler 46, 50, 116, 304, 305, 319, 320, 325, 327, 328, 330, 350, 353, 471
 - linker 189, 479
 - shell 70, 82, 85, 104, 106, 108, 141, 393, 395, 433
- Error shell call **393**
- errors 301, 471
- EXEC command 81, **140**
- EXEC files 70, 77, 80, 90, 97, **102**, 119, 125, 137, 140, 144, 149, 156, 301, 388, 394, 420, 423, 503
 - parameter passing 81-85, 103, 143, 157
 - redirection 107
- EXEC language 81, 102, 140
- executable files 35, 41-43, 51, 72, 96, 97, 105, 112, 183-185, 187, 304, 322, 331, 489, 497
- EXECUTE command 84, 104, 105, **140**, 157
- Execute shell call **394**
- executing a program 119, 123
- EXISTS command **140**
- EXIT command **141**
- EXPAND directive 46, 47, 303, **327**, 344, 475, 478
- ExpandDevices shell call **396**
- EXPORT command 83, 84, 104, **141**, 157
- Export shell call **398**
- EXPRESS command **141**
- expressed files 148, 186
- expressions
 - assembler **309-311**, 321, 348, 461, 473, 474, 478, 481
 - assembly language 25, 28
 - conditional assembly language 353
 - macros 63
 - object module format 493, 494, 496
 - shell 86, **106**, 145
- ExpressLoad 141
- extended files (see resources)

Appendices

external labels 90, 184, 185, 304, 309, 474, 493

Eyes, David xxv

F

FastFile shell call **400**

file access flags 122, 131, 138, 142

file length 122

file names 35, **100**, 121, 302, 507

file not found error 1, 401

file system translator 146

file types 35, 102, 104, 105, 121, 142, 150, 151, 159, 183, 483, 484, 487, 497

 auxtype (see auxiliary file type)

 SRC 102

 TXT 102

files 35

FILETYPE command 73, **142**

Finder 69, 73, 503

Fischer, Michael xxv

fixed address 331

floating-point data **323**, 328, 330, 476

 double precision **324**, 476

 extended precision **324**

FOR command 84-87, 103, 119, 125, 139, **143**

formatted I/O 382

formatting disks **139**, **145**

FST 146

G

game paddles 443, 463

GBLA directive 63, 66, 340, **350**

GBLB directive 63, 340, **350**

GBLC directive 63, 340, **350**

GEN directive 61, 149, 303, 336, 344, **350**, 475, 478

GEQU directive 23, 27, 31, 303, 309, 312, **327**, 332, 345, 472, 473, 474, 475

GET4 macro 53

GetCommand shell call **405**

GetIODevices shell call **407**

GetLang shell call **408**

GetLInfo shell call 388, **409**, 432, 433

GetLInfoGS macro 72

GETx macro 40, **377**

GOTOXY macro **379**

GS/OS xxv, 106, 507

 errors 471

GS/OS aware 99

GS/OS input strings 447

GS/OS macros 356

GS/OS output strings 448

GS/OS strings 386

GSBug

 altering memory 282

 breakpoints 252, 255, 259, 274, 275

 BRK 232

 command filters 251, 258

 command line 242, 279-291

 configuration file 232, 235, 286, 287

 debugging 243, 246, 250, 254, 266, 271

 direct page 259

 displaying memory 240, 242, 243, 272, 280

 expressions 285

 GS/OS calls 270

 hexadecimal 284

 installation 231

 master display 239

 memory display 240

 memory protection 251, 256, 276

 mini-assembler 268

 printing 243

 quitting 244

 registers 257, 259, 260, 261, 262, 263, 265

 segments 253

 selecting displays 237, 239

 stack 260, 265

 toolbox calls 270

 trace history 249, 252

GSBug Debugger 229-291

H

hard disks 501

hard reference **322**, 494

header files 503

HELP command 7, 9, 40, 55, 96, 99, **144**, 503

hexadecimal constant 314, 476

hexadecimal data **322**

hidden characters 163

High Sierra 146

Appendix E: Differences Between ORCA/M 2.0 and ORCA/M 1.0

HISTORY commands **144**

HOME command **145**

HOME macro **380**

I

I/O redirection (See redirection), 508

IBM 370 assembler 24

IEEE directive 303, 323, **328**, 344, 475, 476, 478

IF command 86, 87, 139, **145**, 149

immediate addressing 309, 310, 329, 359

IN clause of FOR command 143

INC2 macro **451**

INC4 macro **451**

indexed addressing 310

indirect addressing 359

INIT command 12, **145**

INIT utility 99

Initialization Programs 75

Inits 75

InitWildcard shell call **414**

INPUT command **147**

installing ORCA/M 499

INSTIME directive 303, **328**, 344, 478

instruction set 50

integer data **321**

J

Jcn macros **452**

jump table 496

K

KEEP directive 23, 24, 26, 105, 117, 184, 191, 302, 303, **328**, 336, 411, 435, 475, 479, 483

keep name 187

KEEP parameter 105, 117, 148, 152, 184, 191, 302, 411, 435, 483

keyboard buffer 438

KIND directive **328**, 475

L

LA macro **453**

labels 24, 47, 65, 474, 478, 481-483, 490, 492, 496

global 30-31, 41, 43, 90, 117, 184-186,

189, 301, 304, 325-327, 332, 333

local 27, 31, 304, 319, 325, 326, 333, 334, 472, 478, 496

macros 54, 60, 61

syntax **308**, 317

Labiak, William xxv

language card 491

language names 81, 107, 122, 140, 154, 157, 159

language numbers **96**, 157, 179, 406, 408, 431

language stamp 77, 80, 81, 96, 102, 116, 122, 123, 138, 140, 300

Languages prefix 39, 97, 99, 112

LCLA directive 62, 65, 66, 340, 343, 348, **351**

LCLB directive 62, 340, **351**

LCLC directive 62, 66, 340, 341, 346, **351**

LENGTH attribute 66, **344**, 478, 494, 495

libraries 42, 98, 105, 134, 148, 151, 152, 183-186, 301, 327, 361, 481, 501, 503, 505

creating **191**

Libraries prefix 39, 43, 97, 98, 185, 191

library files 35

Lichty, Ron xxv

line editor 9, 89, **93-95**, 388

clear-line command 94

cursor-left command 94

cursor-right command 94

delete-character-left command 94

delete-to-end-of-line command 94

end-of-line command 94

execute command 95

insert mode 94

start-of-line command 94

line length

assembly language 307

LINK command 11, 41, 42, 86, 102, 105, 119, **147**, 184, 192, 302, 412, 434

link editor (See linker)

link map 42, 117, 147, 186, 301

linker 42, 90, 98, 105, 112, 118, 147, **183-193**, 304, 322, 331-333, 401, 434, 468, 474, 503

output **188**

output name 187

script files 148, 192

Appendices

LINKER command **148**

linking a program 42, 116, 119, 123, 184, 299, 318, 322

Lisa 146

LIST directive 46, 47, 117, 301, 303, 327, **329**, 336, 344, 475, 478

LLA macro **454**

LM macro **455**

load files 147

See also executable files

load module 185

load segment 26, 48, 150, 184, 189, 190, 318, 320, 332, 333, 481, 491, 492, 493

loader 43, 72, 90, 183, 189, 191, 325, 357

Loader Dumper 291-298

location counter 28, 312, 319, 326, 331, 480, 494, 495, 496

locking files 131

logical operators

assembly language 311

LOGIN file 39, 82, 84, 99, 116, 141, 154, 157, 502

long addressing 28, 31, 191, 309, 310, 320, 327, 454, 472

long command lines 102

long indirect addressing 359

LONG macro 50, **456**

LONGA directive 50, 303, **329**, 344, 456, 467, 475, 478

LONGI directive 50, 303, **329**, 344, 456, 467, 475, 478

LOOP command 119, 125, 139, **149**

LRET **469**

LSR2 macro **457**

LSUB macro 457, **469**

M

M16.GS.OS file 356

M16.I.O file 373

M16.SHELL file 385

M16.TOOLS file 357

MacGen utility 3, 55, 56, **149**, 355

Macintosh 146

macro buffer 335, 351, 352

MACRO directive 59, 66, 335, 341, 343, 347, **351**, 472, 474, 478

macro files 149

macro model line 59, 62

macros 505

calling 337, 339, 355, 472

expansion 55, 61, 67, 299, 305, 332, 336, 341, 350, 352

keyword parameters 57, **339**, 472, 477

libraries 53, 56

macro model statements 59

model statements 337, 474

naming 59

nesting 475

parameter passing 60, 66

parameters 57, 59

positional parameters 57, 472, 477

using 53-57

writing 59-67, **335-346**

MAKEBIN utility **150**

MAKELIB command 191

MAKELIB utility 98, 99, **151**, 183

MASL macro **458**

MCOPY directive 55, 150, 299, 303, 335, 336, **351**, 355, 474, 475, 477, 478, 479

MDROP directive 55, 303, **352**, 474, 475, 477

MEM directive 303, **329**

memory management 48, 51, 72, 318, 331, 361, 479, 494

Memory Manager 69, 401, 415, 422, 428

memory model 503

MEND directive 59, 62, 66, 335, 337, 341, 343, 347, **352**, 472, 475, 478

merging files 160

MERR directive 50, 116, 303, **330**, 475

MEXIT directive **352**, 478

MLOAD directive 299, 303, 335, **352**, 474, 475, 477, 479

MLSR macro **458**

MNOTE directive 346, **353**

modification date 153, 159

modify time 122

MODx macro **367**

mouse 167

MOVE command 11, 101, **152**, 191

MOVE macro **459**, **460**

MS/DOS 146

MSB directive 30, 47, 303, 323, **330**, 344, 475, 478

MUL4 macro 53

multiple languages 3, 12

MULx macro **368**

Appendix E: Differences Between ORCA/M 2.0 and ORCA/M 1.0

N

native mode 69, 136, 480
NDA 75
networks 98
New Desk Accessory 75
NEWER command **153**
NextWildcard shell call **417**
NORMCH macro **380**
NOT operator 312
null driver 509
NUMSEX directive 303, 324, **330**, 344, 475, 478

O

OBJ directive **330**, 473
OBJCASE directive 303, **330**, 344, 478
object module format 3, 133, 135, 152, 484, 487, 490
object modules 35, 41, 42, 90, 105, 117, 119, 125, 128, 147, 151, 183-186, 301, 322, 328, 333, 400, 401, 411, 435, 468, 475, 479, 484, 487
object segment 42, 118, 137, 147, 186, 189, 301, 303, 304, 328, 480, 487, 491
object types 50
OBJEND directive **331**
octal constant 314, 476
OMF 124
 see also object module format
OMF 1.0 512
OMF 2.0 512
on-line help 73
operands
 assembly language 25, 28, 309
 macros 54, 57, 60
operation codes
 assembly language 25, 299, 308, 332, 475
 macros 54, 335
 object module format 491, 497
ORCA disk 8, 10, 11
ORCA macros 359
ORCA string 360, 382
ORCA system hierarchy 89
ORCA.Extras disk 56, 355
ORCA.Sys16 file 11, 501
ORCA/C language 96

ORCA/Pascal language 96
ORG directive 51, 318, **331**, 345, 473, 475, 482
output (See standard output, redirection)

P

page boundary 318, 328, 480
partial assembly 41, 86, 117, 128, 185, 300, 302, 410, 412, 433, 473
Pascal 97, 146, 469, 501
Pascal protocol 70, 360
Pascal strings 450
path names 26, 36, 45, 79, 97, 102, 106, 118, 121, 396, 409, 414, 432, 479, 507
PHx macros **461**
pipes 90, 98, **111**, 502
PLx macros **462**
PopVariables shell call **419**
PRBL macro **381**
PREAD macro **463**
PREFIX command 2, 10, 39, 97, 98, **154**
prefixes 26
 0-7 39, 99
 numbers 38, **97**, 99, 100, 112, 396, 414, 508
 standard 37, 38
PRINTER directive 46, 47, 119, 303, 306, **332**, 344, 475, 478
printer driver 505, 509
printers **108-111**, 505, 509
 characters per line 109
 configuration 108
 control characters 110
 lines per page 109
 redirecting output 107, 108
 slot 109
printing files 107, 160
PRIVATE directive 25, 48, 118, 190, 303, 304, 328, **332**, 474
PRIVDATA directive 25, 48, 118, 190, 303, 304, 328, **332**, 474
PRIZM 502
ProDOS 146, 507
ProDOS 16 xxv
ProDOS 8 xxv, 39, 48, 72, 97, 142, 150, 318, 324, 325
PRODOS command 96, **154**, 159
PRODOS language 96, 102, 154

Appendices

- ProDOS macros 356
- program launchers 97
- program segmentation 27, 48, **189**
- program size 122
- programming examples 2, 31, 32, 48, 54, 60, 61, 70, 81, 85, 86, 87, 119, 125, 144, 145, 149, 190, 336
- progress information 301
- prompt (see shell prompt)
- purging files 400, 401
- PushVariables shell call **420**
- PUT4 macro 53, 57
- PUTCR macro 53, **384**
- PUTS macro 53, 406
- PUTx macro **382**

Q

- QUIT command **154**

R

- RAM 37, 39, 98
- RAM disks 501
- random number generator 466
- RANx macro **369**
- Read shell call 427
- ReadIndexed shell call **421**, 427
- ReadVariable shell call **423**
- Redirect shell call **425**
- redirection 40, 70, 79, 82, 90, 96, 107, 119, 306, 377, 388, 390, 391, 425, 508
- relocation 43, 51, 72, 90, 183, 304, 309, 328, 331, 481, 490
- relocation dictionary 48, 184, 492, 493, 497
- RENAME command 11, 95, 101, **155**, 415
- RENAME directive 303, **332**, 475, 478
- RESEQUAL command **155**
- resource description file 196, 198
- resource fork 122, 126, 226
- resources 126, 129, 155, 156, 195-228, 511
 - see also Rez
- restartability 112, 124, 405
- restartable programs 96
- RESTORE macro **464**
- RET macro 456, 464, **469**
- return characters 163
- Rez 129, 195-228
 - append statement 220

- arrays 207, 210, 213
- built-in functions 213, 222
- change statement 209
- comments 196
- conditional compilation 216
- data statement 200
- define statement 216
- delete statement 209
- escape characters 225
- expressions 221
- if statement 216
- include statement 98, 198, 219
- labels 212, 213, 214, 215
- macros 216
- options 227
- printf statement 217
- read statement 200
- resource attributes 199
- resource statement 210, 220-226
- resource types 201
- strings 204, 224, 225
- switch statement 208, 210
- symbols 209
- type statement 201
- undef statement 216
- variables 222

- REZ command **156**

- ROM code 51, 318, 330, 331

- ROM disks 501

- RUN command 5, 11, 41-43, 55, 105, 116, **156**, 184, 299, 300, 330

S

- S16 files 97

- S16 programs 73

- SANE Tool 323, 324, 330

- SAVE macro **465**

- Scanlon, Leo J. xxv

- script files

 - linker 148

- scripts

 - see EXEC files 502

- SEED macro **466**

- segment type 328, 489

- segmentation (See program segmentation)

- segments 189, 291

- separate compilation 43, 185, 332

Appendix E: Differences Between ORCA/M 2.0 and ORCA/M 1.0

- sequence symbols 26, 65, 308, **343**, 347, 348, 477, 482
- SET command 79, 81, 83, 84, 85, 86, 87, 103, 149, **156**, 160
- Set shell call **427**
- SETA directive 63, 65, 66, 340, 343, 348, **353**, 477
- SETB directive 63, 340, **353**
- SETC directive 63, 66, 340, 341, 346, **354**, 477
- SETCOM directive 46, 47, 303, **333**, 474, 475
- SetErrorDevice call 430
- SetInputDevice call 430
- SetIODevices shell call 407, **429**
- SetLang shell call **431**
- SetLInfo shell call 409, 410, **432**
- SetOutputDevice call 429
- SetStopFlag shell call **437**
- setting attribute **344**, 478
- shell 501
 - calling **385**
 - command table 501
 - parameter passing 432
- shell commands 1, 77, 89, 102, 408
 - built-in commands 12, 96, 112
 - command expansion 9, 94
 - command list 94, 95
 - command parameters 41
 - command types 12, **95**, 112
 - entering 9
 - language commands 12
 - language names **96**, 112
 - metacharacters 118
 - multiple commands 95, 103
 - parameters 12, 116, 140
 - utilities 96
 - utility commands 12, 112
- shell identifier 70
- shell prefix 97, 112, 124, 170, 509
- shell prompt 1, 95, 105
- shell variables 78, 81, **103**, 104, 140, 147, 419, 423, 509
 - assigning values to 79, 103, 156, 160, 427, 439
 - expansion 79
 - metacharacters 105, 302
 - scope 83, 84, 104, 141, 157, 394, 398, 420, 421
 - string concatenation 79
- SHORT macro 50, **467**
- SHOW command 9, 37, 100, 107, 123, 154, **157**
- SHUTDOWN command **157**
- SIGNx macro **370**
- soft reference **322**, 468, 495
- SOFTCALL macro **468**
- SOS 146
- source files 35, 159, 183, 191, 299, 307, 487
- sparse files 122
- SQRTx macro **371**
- stack 361, 461, 462, 464, 465
- stack addressing 311
- stack frames 469
- stack size 509
- standard input 69, 97, 107, 348, 425, 430, 438, 508
- standard output 40, 69, 97, 107, 120, 133, 137, 159, 359, 374-376, 379-382, 384, 391, 394, 414, 425, 429, 508
- standard prefixes **97**, 99, 154, 157
- START directive 23, 25, 26, 30, 48, 118, 190, 303, 304, 318, 320, 328, 331, 332, **333**, 336, 474
- static segments 150, 489, 490, 493, 494
- static variables 332, 494
- status register 50
- Stop shell call 437, **438**
- storage type 122
- strings 323
- strings, ORCA macros 449
- SUB macro **469**
- SUB4 macro 53
- SUBx macro **372**
- SWITCH command 125, **158**
- SYMBOL directive 46, 47, 301, 303, 304, **333**, 344, 475, 478
- symbol tables
 - assembler 41, 46, 117, 301, 304, 326, 327, 333
 - linker 42, 117, 147, 184, 186, 189, 301, 483
- symbolic parameters 60, 307, **337**, 472, 474, 477, 478
 - arrays of 63, 65, **339**, 473, 477
 - assigning values to 63, 337, 340, 348, 349, 353, 354, 472, 477

Appendices

- booleans 62, 63, 67, 337, 345, 350, 351, 353
- concatenation 341
- defining 62
- dot operator 341
- integers 62, 63, 64, 67, 337, 345, 349, 350, 351, 353
- positional parameters 337, 338
- scope 340, 350, 351
- strings 60, 62, 63, 64, 67, 337, 345, 348, 349, 350, 351, 354
- SYSCMND file 39, 72, 99, 112, 124, 406, 501
- SYSEMAC file 39, 99, 165, 502
- SYSHELP file 170, 502
- SYSLIB 505
- SYSPIPEX files 111
- SYSTABS file 39, 98, 99, 179, 502
- system configuration 499, 501
- SYSTEM files 97
- System prefix 39, 72, 509
- system programs 69, 73
- system requirements xxiv
- SYSTEMP file 20, 168, 172

T

- tabs 163, 166, 174, 180
 - ENTAB utility 139
 - removing 160
- terminal errors 117, 301
 - assembler 305, 478
 - linker 483
- TEXT command **159**
- text files 96, **102**, 112, 154, 159, 160, 355, 487
- TEXT language 96, 102, 159
- Text Tools 40, 59, 69, 70, 335, 373, 377, 388, 390, 429
- time 122
- TITLE directive 47, 304, **334**, 475
- Toolbox, Apple IIGS xxv, 29, 48, 55, 59, 70, 82, 335, 356, 357, 373, 393, 422, 426, 428
- tool errors 471
- TOUCH command **159**
- TRACE directive 64, 149, 304, 308, 340, 343, 344, 346, 348, **354**, 475, 478
- two's complement notation 360

- type attribute **345**, 478, 494, 495

- TYPE command 8, 139, **159**

- type conversions 444

U

- UNALIAS command 116, **160**

- unlocking files 138

- UNSET command 80, 83, 85, 86, 103, 104, 149, 157, **160**, 439

- UnsetVariable shell command **439**

- user ID 70, 72

- USING directive 31, 320, **334**, 481, 483

- utilities 81, 497, 503

- Utilities prefix 39, 72, 96, 97, 99, 112, 144

V

- variables (See assembler variables, shell variables, symbolic parameters, DC directive, DS directive, ANOP directive, labels)

- assembler 320

- version

- editor 167

- version numbers 440, 484, 490

- Version shell call **440**

- volume names 2, 36, 100, 120, 125, 126, 139, 145, 157

W

- wait flag 41, 117, 301, 304

- wildcards 11, **101**, 121, 126, 128, 138, 140, 152, 158, 159, 414, 417

- word size 50, 69, 328, 329, 344, 357, 388, 456, 467, 480

- work prefix 20, 39, 97, 98, 111, 168, 172

- write protect 127, 483