 Macintosh®

---

**Macintosh Programmer's  
Workshop 3.0 Assembler  
Reference**

---

• APPLE COMPUTER, INC.

This manual and the software described in it are copyrighted, with all rights reserved. Under the copyright laws, this manual or the software may not be copied, in whole or part, without written consent of Apple, except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format.

You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

© 1985-88 Apple Computer, Inc.  
20525 Mariani Ave.  
Cupertino, California 95014  
(408) 996-1010

Pascal Compiler © 1982-88  
Apple Computer, Inc.  
© 1981 SVS, Inc.

Apple, the Apple logo, AppleShare, AppleTalk, A/UX, ImageWriter, LaserWriter, Lisa, MacApp, Macintosh, and SANE, are registered trademarks of Apple Computer, Inc.

MPW, QuickDraw, ResEdit, APDA, and SADE are trademarks of Apple Computer, Inc.

MacDraw, MacPaint, and MacWrite are registered trademarks of Claris Corporation.

Microsoft Word is a trademark of Microsoft Corporation.

ITC Garamond and ITC Zapf Dingbats are registered trademarks of International Typeface Corporation.

POSTSCRIPT is a registered trademark of the Adobe Systems Incorporated.

Adobe Illustrator 88 is a trademark of Adobe Systems Incorporated.

ImageStudio is a trademark of Esselte Pendaflex Corporation in the United States, of LetraSet Canada Limited in Canada, and of Esselte LetraSet Limited elsewhere.

Motorola is a trademark of Motorola, Inc.

MathType is a trademark of Design Science, Inc.

QMS is a registered trademark of QMS, Inc.

UNIX is a trademark of AT&T Bell Laboratories.

Simultaneously published in the United States and Canada.

MPW sample programs

Apple Computer, Inc. grants users of the *Macintosh Programmer's Workshop* a royalty-free license to incorporate *Macintosh Programmer's Workshop sample programs* into their own programs, or to modify the sample programs for use in their own programs, provided such use is exclusively on Apple computers. For any modified *Macintosh Programmer's Workshop* sample program, you may add your own copyright notice alongside the Apple copyright notice.

# Contents

Figures and tables xiii

## **Preface About This Manual xv**

What this manual contains xvii

Other reference materials xviii

Notation conventions xix

Aids to understanding xix

    Courier typeface xx

    Italic xx

    Fields xxi

    Delimiter symbols xxi

    Braces xxii

    Brackets xxii

    Ellipses xxiii

    Underlining xxiii

For more information xxiv

## **Part I Using the Assembler 1**

### **1 About the Assembler 3**

    General characteristics 5

    Overview of the assembly process 6

    Assembly files 7

    Programming for the Macintosh 8

    Macintosh libraries 9

### **2 Coding Conventions 11**

    Source text structure 13

        Scope of definitions 15

            Imported and exported objects 17

            @-labels 17

            Summary 18

        Segmentation 18

Machine instruction syntax	19
The label field	20
The operation field	21
The operand field	23
Comments	24
Symbols	25
Identifiers	25
Numeric constants	26
Strings	27
Expressions	28
Evaluation of expressions	30
Absolute and relocatable expressions	31
Absolute expressions	32
Relocatable expressions	33

### **3 Address Syntax 35**

Addressing modes	37
Ambiguities and optimizations	41
Forward-reference addressing	43
Registers	44
Special address formats	46
MC68xxx instructions	46
MOVEM: Multiple moves	46
MC68020 instructions	47
MULS and MULU: Signed and unsigned multiplication	47
DIVS and DIVU: Signed and unsigned division	47
TDIVS and TDIVU: Truncated signed and unsigned division	47
PACK and UNPK: Packing and unpacking	48
CAS and CAS2: Comparing and swapping	48
Bit field instructions	49
Tcc and TPcc: Trap on condition	49
Assembler control	49
The MC68030 processor	49
Assembler control	50
MC68020 statements you can use	50
MC68851 instructions you can use	50

MC68881 and MC68882 instructions	52
FMOVM with explicit register lists	52
FMOVE with packed BCD data	52
FSINCOS: Simultaneous sine and cosine	53
FTcc and FTPcc: Floating-point trap on condition	53
FTEST: Text operand and set floating-point condition codes	53
MC68851 instructions	53
Literals	55

#### **4 Assembler Directives 57**

Assembler directives	59
Code and data module definitions	59
Symbol definitions	59
Data definitions	59
Template definitions	59
Linker and scope controls	60
Assembly options	60
Location-counter controls	60
File controls	60
Listing controls	60
Directive formats	61
Code and data module definitions	62
PROC and ENDPROC: Define procedure code module	62
FUNC and ENDFUNC: Define function code module	63
MAIN and ENDMAIN: Define main program code module	63
RECORD and ENDR: Define a data module	64
INCREMENT and DECREMENT	65
MAIN	66
CODE and DATA: Switch between code and data	67
END: End the assembly	67
Symbol definitions	68
EQU and SET: Name constants and registers	68
REG and FREG: Name register list	70
OPWORD: Name machine instruction	71
Data definitions	72
DC and DCB: Place constants in code or data	73
DS: Define storage area	75

- Template definitions 76
  - RECORD and ENDR: Define a template 76
    - Using templates as data types 81
  - WITH and ENDWITH: Supply RECORD name qualification 82
- Linker and scope controls 84
  - EXPORT and ENTRY: Expand scope of entry points 85
  - IMPORT: Identify external entry points 87
  - CODEREFS and DATAREFS: Control name linking 88
    - Code-to-code references 89
    - Code-to-data references 90
    - Data-to-code references 90
    - Data-to-data references 91
  - SEG: Specify current code segment 92
  - COMMENT: Place a comment in object file 93
- Assembly options 93
  - MACHINE: Specify target machine 93
  - MC68881: Assemble MC68881/MC68882 coprocessor instructions 94
  - MC68851: Assemble MC68851 coprocessor instructions 95
  - STRING: Specify string format 95
  - BRANCH and FORWARD: Resolve forward branches 96
  - OPT: Specify level of code optimization 97
  - CASE: Specify treatment of lowercase letters 98
    - Writing register names 99
  - BLANKS: Control acceptance of blanks in operand field 99
- Location-counter controls 100
  - ALIGN: Align location counter 100
    - Special cases 101
  - ORG: Set location counter 102
- File controls 103
  - File search rules 104
  - INCLUDE: Take source text from another file 104
  - DUMP and LOAD: Write and read symbol table files 105
  - ERRLOG: Specify error log file 106
- Listing controls 107
  - PAGESIZE: Specify listing page size 107
  - TITLE: Specify title line for listing 108
  - PRINT: Control listing information 108
  - EJECT: Start new listing page 111
  - SPACE: Insert blank line in listing 111

## Part II The Macro Processor and the Macro Language 113

### 5 Macros 115

- Macro expansion 117
- Scope of macro symbols 118
- Defining macros 118
  - MACRO and ENDM or MEND: Delimit macro 119
  - The prototype statement 119
  - The macro body 120
  - Macro comments 121
  - Symbolic parameters 123
    - Concatenating symbolic parameters 124
- Calling macros 125
  - The macro-qualifier 126
  - Macro call labels 127
  - Operand syntax 128
    - Paired single quotation marks 128
    - Paired parentheses and brackets 128
    - Ampersands 129
    - Commas 129
    - Blanks (spaces and tabs) 129
    - Backquotes 130
    - @-labels 130
    - Omitted or extra operands 130
  - Operand sublists 131
    - Accessing sublist elements 131
    - Parameter types and default values 132
  - Nesting macros 133
- Keyword macros 135
  - Defining keyword macros 135
  - Calling keyword macros 136
- Mixed-mode macros 138

### 6 Macro Variables and Functions 139

- SET variables 141
  - SET variables and symbolic parameters 143
  - LCLA, LCLC, GBLA, and GBLC: Define SET variables 143

- SETA and integer expressions 145
  - &ABS: Return absolute value 146
  - &EVAL: Evaluate contents of string 147
  - &ISINT: Test string for integer content 147
  - &LEN: Measure string length 147
  - &LEX: Parse string lexically 148
  - &LIST: Divide string into list 150
  - &MAX: Find maximum in integer list 151
  - &MIN: Find minimum in integer list 151
  - &NBR: Count sublist elements 151
  - &ORD: Return integer value 152
  - &POS: Find position of substring in string 152
  - &SCANEQ and &SCANNE: Scan string 153
  - &STRTOINT or &S2I: Convert string to integer 154
- Symbol table functions 154
  - &NEWSYMTBL: Create new symbol table 154
  - &ENTERSYM: Enter or update symbol in table 155
  - &FINDSYM: Find symbol in table 156
  - &DELSYMTBL: Delete symbol table 157
- SETC and string expressions 157
  - Accessing substrings of string variables 158
  - &CHR: Convert integer to character 159
  - &CONCAT: Concatenate strings 160
  - &DEFAULT: Return string value or default 160
  - &GETENV: Return MPW Shell variable value 160
  - &INTTOSTR or &I2S: Convert integer to string 160
  - &LOWCASE or &LC: Convert string to lowercase 161
  - &SETTING: Return directive setting 161
  - &SUBSTR: Return substring of string 162
  - &TRIM: Trim spaces and tabs from string 163
  - &TYPE: Determine identifier type 163
  - &UPCASE or &UC: Convert string to uppercase 164
- SET array variables 165
  - Defining SET array variables 165
  - Using SET array variables 166
  - Accessing substrings in SET array string elements 167



Assembler system variables	168
&SYSINDEX or &SYSNDX: Macro call index	168
&SYSLIST or &SYSLST: Macro operand list	169
&SYSSEG: Current segment identifier	170
&SYSMOD: Current module identifier	170
&SYSDATE: Current date	170
&SYSTIME: Current time	170
&SYSTOKEN and &SYSTOKSTR: Values set by &LEX	171
&SYSVALUE and &SYSFLAGS: Values set by &FINDSYM	171
&SYSLOCAL and &SYSGLOBAL: System symbol table ID's	171

## **7 Macro and Conditional-Assembly Directives 173**

Boolean control expressions	175
Comparing two integer expressions	175
Comparing two string expressions	175
Comparing integer and string expressions	176
GOTO, IF...GOTO, and macro labels: Branching	176
IF, ELSEIF, ELSE, and ENDIF: Conditional assembly	178
WHILE and ENDWHILE: Looping	179
CYCLE and LEAVE directives	180
ACTR: Limit looping	180
EXITM or MEXIT: Exit macro	181
WRITE and WRITELN: Write to diagnostic output file	181
AERROR: Error generation	182
ANOP: Assembler NOP	182

## **Part III Appendixes 183**

### **A Generic Instruction Formats 185**

### **B Syntax Diagrams 189**

Assembly-language addresses	191
Addressing modes	191
Address optimizations	192

Special address formats	192
MC68000 instructions	192
MOVEM: Multiple moves	192
MC68020 instructions	192
MULS and MULU: Signed and unsigned multiplication	192
DIVS and DIVU: Signed and unsigned division	193
TDIVS and TDIVU: Truncated signed and unsigned division	193
PACK and UNPK: Packing and unpacking	193
CAS and CAS2: Comparing and swapping	193
Bit field instructions	193
Tcc and TPcc: Trap on condition	193
MC68881 and MC68882 instructions	194
FMOVEM with explicit register lists	194
FMOVE with packed BCD data	194
FSINCOS: Simultaneous sine and cosine	194
FTcc and FTPcc: Floating-point trap on condition	194
FTEST: Test operand and set floating-point condition codes	194
MC68851 instructions	195
Literals	195
General assembly directives	196
Macro and SET variable directives	200
SET variable functions	202

## **C Assembly Listing Format 205**

## **D Other Assemblers 211**

Syntax comparison	213
Writing identifiers	213
Writing numbers	214
Writing strings	214
Defining modules	215
Communicating between modules	215
Writing expressions	215
Location-counter reference	216
Addressing features	217
Writing macros	217

**E The Macintosh Character Set 219**

**F Instruction Sets 223**

- Instruction evaluation 225
- Listing conventions 225
  - Opcode 226
  - Operands 226
  - Opcode word 227
  - Cp type 228
  - Group 228
  - Flags 228
  - Range 229
  - Equivalent 229
- Condition codes 229
- Instruction set listings 233

**G Assembler Command Syntax 253**

- Assembler command syntax 255

**H Object Assembler Macros 261**

- InitObjects 263
- ObjectDef 263
- ObjectIntf and the IMPL keyword 265
- ObjectWith and EndObjectWith 266
- ProcMethOf, FuncMethOf, and EndMethod 267
- MethCall 268
- Inherited 268
- NewObject 269
- MoveSelf 269

**I Pascal and C Calling Conventions 271**

- Pascal calling conventions 273
  - Parameters 273
    - Real-type parameters 274
    - Structured-type parameters 275
  - Function results 275
  - Register conventions 277

C calling conventions 277  
Parameters 278  
Function results 278  
Register conventions 278

## **J Structured Assembly Macros 279**

Structured macro statements 281  
Expressions 281  
Flow-control macros 283  
The If statement 283  
The Switch statement 285  
The Repeat statement 287  
The While statement 287  
The For statement 288  
The Leave statement 290  
The Cycle statement 291  
The GoTo statement 292  
Program structure macros 292  
Sample code generation from program structure macros 294  
Procedure and function header 295  
Local variable declaration 298  
Procedure or function start 299  
Procedure or function secondary entry point 300  
Procedure or function exit 301  
Procedure, function, or trap invocation 303  
Considerations for use 306  
Why you should or should not use the structured assembly macros 307  
Rules for using structured assembly macros 308  
Syntax summary 309  
Expressions 309  
Flow-control macros 310  
Program structure macros 311

**Glossary 313**

**Index 319**

# Figures and tables

## Part I Using the Assembler 1

### 1 About the Assembler 3

Table 1-1 Assembler status codes 7

### 2 Coding Conventions 11

Figure 2-1 Source text structure 14

Table 2-1 Data size qualifiers 22

Table 2-2 Operators 29

### 3 Address Syntax 35

Table 3-1 Address symbols 37

Table 3-2 Address syntax summary 38

Table 3-3 Effective address transformations 42

Table 3-4 Registers 44

Table 3-5 MC68851 registers in the MC68030 50

Table 3-6 MC68851 instructions valid for the MC68030 51

Table 3-7 Special MC68851 operand formats 54

### 4 Assembler Directives 57

Figure 4-1 Stack frame example 79

Figure 4-2 Sample template format 103

Table 4-1 DC and DCB data increments 73

Table 4-2 Effects of CODEREFS and DATAREFS 91

Table 4-3 PRINT directive parameters 109

## Part II The Macro Processor and the Macro Language 113

### 6 Macro Variables and Functions 139

Table 6-1 Values returned by &LEX 148

Table 6-2 &SETTING values 162

Table 6-3 Assembler system variables 168

## **Part III Appendixes 183**

### **A Generic Instruction Formats 185**

Table A-1 Generic instruction conversions 187

### **C Assembly Listing Format 205**

Figure C-1 Default assembly listing format 207

### **D Other Assemblers 211**

Table D-1 Identifier syntax rules 213

Table D-2 Number syntax 214

Table D-3 String syntax 214

Table D-4 Module definition 215

Table D-5 Communication directives 215

Table D-6 Allowable operators 216

Table D-7 Addressing features 217

### **F Instruction Sets 223**

Table F-1 Instruction operands 227

Table F-2 MC68xxx condition codes 230

Table F-3 MC68881 IEEE nonaware tests 231

Table F-4 MC68881 IEEE aware tests 231

Table F-5 MC68881 miscellaneous tests 232

Table F-6 MC68851 PMMU condition codes 232

Table F-7 MC68000, MC68010, and MC68020/MC68030 instructions 233

Table F-8 MC68881 instructions 243

Table F-9 MC68851 instructions 249

### **I Pascal and C Calling Conventions 271**

Table I-1 Parameter passing conventions 273

Table I-2 Function-result passing conventions 276

## Preface About This Manual

THIS MANUAL TELLS YOU HOW TO PREPARE SOURCE FILES to be assembled by the Macintosh® Programmer's Workshop Assembler (also called the MPW Assembler).

This manual assumes that you are generally familiar with assembly-language programming. It also assumes that you understand and are able to write the symbolic assembly language for the Motorola MC68xxx instructions you want to use. ■

### *Contents*

What this manual contains	xvii
Other reference materials	xviii
Notation conventions	xix
Aids to understanding	xix
Courier typeface	xx
Italic	xx
Fields	xxi
Delimiter symbols	xxi
Braces	xxii
Brackets	xxii
Ellipses	xxiii
Underlining	xxiii
For more information	xxiv





---

## What this manual contains

This manual is divided into 7 chapters in two sections, and 10 appendixes. Here is a summary of the information it contains:

- Part I is about using the Assembler. It contains 4 chapters.
  - Chapter 1, "About the Assembler," lists some characteristics of the Macintosh Workshop Assembler and describes its general mode of operation. It also includes a summary of file-naming conventions.
  - Chapter 2, "Coding Conventions," discusses the overall structure of MPW assembly-language source text. It includes information about statement and directive formats, symbol formation, and the evaluation of expressions.
  - Chapter 3, "Address Syntax," describes the ways you can address the Macintosh memory and gives the syntax rules for writing addresses in your source text.
  - Chapter 4, "Assembler Directives," provides detailed instructions for using most of the MPW Assembler directives, grouped by the kinds of tasks the macros perform. Macro-expansion directives are covered in Chapter 7.
- Part II covers the Macro Processor and the Macro Language features that relate to it. It contains 3 chapters.
  - Chapter 5, "Macros," tells you how to define and call macros in your source text.
  - Chapter 6, "Macro Variables and Functions," tells you how to use SET variables to program the expansion of your macros.
  - Chapter 7, "Macro and Conditional-Assembly Directives," describes the directives of the MPW Assembler macro language.
- Part III contains the appendixes.
  - Appendix A, "Generic Instruction Formats," gives you the rules for writing the generic forms of some assembly-language statements that the MPW Assembler accepts and converts to specific instructions.
  - Appendix B, "Syntax Diagrams," contains copies of all the syntax diagrams used in this manual.
  - Appendix C, "Assembly Listing Format," describes the way the Assembly listing is constructed.
  - Appendix D, "Other Assemblers," compares the MPW Assembler with other assemblers available for the Macintosh and tells you how to use programs that translate source text from other forms.
  - Appendix E, "The Macintosh Character Set," shows the characters in the Macintosh character set and gives their numeric values.

- Appendix F, "Instruction Sets," lists the MC68000, MC68010, MC68020, MC68030, MC68851, and MC68881/MC68882 machine instructions and condition codes accepted by the Assembler.
- Appendix G, "Assembler Command Syntax," defines the syntax for writing the Assembler command line, including information about the Assembler options.
- Appendix H, "Object Assembler Macros," describes the macros provided for object-oriented programming in the MPW assembly language.
- Appendix I, "Pascal and C Calling Conventions," gives the assembly-language calling conventions for routines written in Pascal and C.
- Appendix J, "Structured Assembly Macros," explains how to use the structured macros that provide MPW Assembler with many of the powerful commands usually found only in the higher-level languages.

At the end of this manual you will find a glossary and an index.

---

## Other reference materials

Before trying to write and assemble a Macintosh assembly-language program, you should read and understand the following books:

- Apple® Computer. *Macintosh Programmer's Workshop 3.0 Reference*. A full description of how to use the Workshop's program preparation tools, including the Assembler.
- Motorola. *M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual*, 6th ed. Prentice-Hall, 1988. The latest comprehensive guide to the MC68000 microprocessor.

In addition, you may find these books helpful:

- Apple Computer. *Inside Macintosh*. Vol. I-III. Reading, Mass. Addison-Wesley, 1985. The complete story of the architecture and operation of the 128K and 512K Macintosh, including details on their ROM routines.
- Apple Computer. *Inside Macintosh*. Vol. IV. Reading, Mass. Addison-Wesley, 1986. Additional and updated material covering the Macintosh and the Macintosh Plus.
- Apple Computer. *Inside Macintosh*. Vol. V. Reading, Mass. Addison-Wesley, 1988. New material covering the Macintosh SE and the Macintosh II.

- Apple Computer. *Apple Numerics Manual*. Second Edition. Reading, Mass. Addison-Wesley, 1986. Describes the Standard Apple Numeric Environment (SANE), which includes extended-precision floating-point arithmetic as specified by IEEE Standard 754. Describes each routine in detail, including boundary conditions and exception handling, and explains how to control the floating-point environment.
- Motorola. *MC68020 32-Bit Microprocessor User's Manual*. Second Edition. A guide to the MC68020 instructions and addressing modes.
- Motorola. *MC68030 Enhanced 32-Bit Microprocessor User's Manual*. A guide to the MC68030 instructions and addressing modes.
- Motorola. *MC68851 Paged Memory Management Unit User's Manual*. A guide to the MC68851 coprocessor, with details of all its instructions.
- Motorola. *MC68881/MC68882 Floating-Point Coprocessor User's Manual*. A guide to the MC68881 and MC68882 coprocessors, with details of all their instructions.

---

## Notation conventions

The discussions in this manual include a number of syntax diagrams and source text examples, designed to help you understand exactly how to write source text structures. Syntax diagrams appear as appropriate in the text and are also gathered together in Appendix B. This section tells you how to interpret the symbols used in the syntax diagrams and examples.

---

## Aids to understanding

Look for these visual cues throughout the manual:

- ▲ **Warning**      Warnings like this indicate potential problems. ▲
- △ **Important**      Text set off in this manner presents important information. △
- ◆ *Note:* Text set off in this manner presents notes, reminders, and hints.

**Computer words and phrases** appear in boldface type when they are introduced. The term is defined in the glossary.

---

## Courier typeface

Anything printed in the *Courier* typeface is a sample of actual source text, as it might be processed by the Assembler. Where the *Courier* typeface occurs in syntax diagrams, it indicates fixed symbols of the MPW assembly language. Such symbols, which must be written in the source text as they are written in the syntax diagram, are sometimes called **terminal symbols**.

- ◆ *Note:* To further distinguish them, directives and machine instructions are printed in capital letters. However, you do not need to use capitals in your source text.

---

## Italic

Generic terms that designate information to be supplied by the programmer are printed in *italic*. They are sometimes called **nonterminal symbols**. Such terms may contain hyphens instead of spaces. The most common ones are shown here:

<i>abs-expr</i>	An absolute expression
<i>arith-expr</i>	A numeric expression
<i>rel-expr</i>	A relocatable expression
<i>expr</i>	An absolute or relocatable expression
<i>string</i>	A string constant
<i>str-expr</i>	A string expression
<i>name</i>	An identifier
<i>label</i>	An identifier used as a label
<i>macro-label</i>	A macro label
<i>filename</i>	A string expression representing a filename
<i>rlist</i>	A MOVEM register list
<i>reg</i>	Any MC68000, MC68010, MC68020, or MC68030 register

Nonterminal symbols not shown in this list are defined where they are used for specific statements.

Where a nonterminal symbol is repeated in a syntax diagram, the repetitions are sometimes distinguished by subscripts, as shown here:

*abs-expr*<sub>1</sub>, *abs-expr*<sub>2</sub>

You write two absolute expressions at this point, separated by a comma. The expressions may be the same or different.

In some syntax diagrams, the connective ::= is used to show the possible values for a nonterminal symbol. For example,

*size* ::= *W* | *L*

indicates that you may write either *W* or *L* for the nonterminal symbol *size*.

References to nonterminal symbols in explanatory text are printed in *italic*, so you won't confuse them with ordinary words or phrases.

---

## Fields

Syntax diagrams distinguish the fields into which source text lines are divided by horizontal spacing, as shown here:

[*macro-label*]          INCLUDE          *filename*

The nonterminal symbol *macro-label*, enclosed in brackets, indicates that you may write an optional macro label in the first field. The terminal symbol INCLUDE indicates that you must write the word *include* in any combination of uppercase and lowercase in the second field. The nonterminal symbol *filename* indicates that you must write a filename in the third field. For further details about fields, see "Machine Instruction Syntax" in Chapter 2.

---

## Delimiter symbols

You must write the following delimiter symbols in your source text exactly as they are shown in the syntax diagrams:

- ,          Comma
- ( )        Parentheses
- .          Period
- \*          Asterisk
- =          Equal sign

Occasionally, required delimiters may look like part of the syntax diagram's punctuation. To prevent confusion in such cases, the required marks are enclosed in single quotation marks. For instance, the expression '{'origin''} means that the word *origin*, enclosed in braces, is required.

---

## Braces

Material within braces represents required items, one of which must be chosen. For instance,

```
{ALPHA  
BETA  
GAMMA}
```

indicates that you must write either *alpha*, *beta*, or *gamma* at that point in your source text.

Notice that the alternatives are written on separate lines. Terms on separate lines always represent expressions with distinct meanings. In some instances, alternate choices have the same effect. Such alternatives that mean the same are separated by a vertical bar (|), as shown:

```
{ON|YES  
OFF|NO}
```

Because all the choices in this example are enclosed in braces, you must choose one line or the other. If you choose the first line, you may write either *on* or *yes*; if you choose the second line, you may write either *off* or *no*.

---

## Brackets

Material within brackets represents optional items. Braces within brackets signify that one of the alternatives must be chosen if the material is to be included at all. Here are some examples:

[ *abs-expr* ]      You may write an absolute expression at this point, or nothing.

INCR[EMENT]      You may write either *incr* or *increment*.

```
{ENTRY  
EXPORT}
```

You need not write anything, but if you do, it must be either *entry* or *export*.

---

## Ellipses

An ellipsis containing three dots (...) indicates repetition of the preceding material. If the material is enclosed in brackets, you don't have to write it at all; if the material is not enclosed in brackets, you must write it at least once. A comma before the ellipsis indicates that repetitions must be separated by commas. You may repeat the material indefinitely, subject to the general length limitation for that particular source text structure. Here are two examples:

*abs-expr*,...      Write one or more absolute expressions separated by commas.  
[ *abs-expr* ],...      Write nothing, or write one or more absolute expressions separated by commas.

Occasionally this notation can be ambiguous, in which case a longer form is used. For example,

`-define name [=value] [, name [=value]]...`

indicates that you may write more *name* or *name=value* groups after the first one, with a comma preceding each one.

An ellipsis containing two dots (..) indicates a scalar range. For example, 0..127 means "0 and 127 and all the intervening numbers."

A sequence of three hyphens (- - -) in sample source text indicates lines of source text not specified by the diagram.

---

## Underlining

An underlined item indicates a default or preset value, which the Assembler will assume if you omit an optional parameter. Here is an example:

[ ENTRY  
EXPORT ]

You need not write anything, but if you write nothing, the Assembler will act as if you had written *entry*.

---

## For more information

APDA™ provides a wide range of technical products and documentation, from Apple and other suppliers, for programmers and developers who work on Apple equipment. (MPW is distributed through APDA.) For information about APDA, contact

APDA

Apple Computer, Inc.  
20525 Mariani Avenue, Mailstop 33-G  
Cupertino, CA 95014-6299

1-800-282-APDA, or 1-800-282-2732

Fax: 408-562-3971

Telex: 171-576

AppleLink: DEV.CHANNELS

If you plan to develop hardware or software products for sale through retail channels, you can get valuable support from Apple Developer Programs. Write to

Apple Developer Programs  
Apple Computer, Inc.  
20525 Mariani Avenue, Mailstop 51-W  
Cupertino, CA 95014-6299



## Part I **Using the Assembler**



## Chapter 1 About the Assembler

THE MPW ASSEMBLER IS CONTAINED IN A SINGLE FILE named `Asm`. This chapter describes some of its general characteristics. For instruction on invoking the Assembler, including further information about the environment in which it runs, see the *Macintosh Programmer's Workshop 3.0 Reference*. A summary of Assembler command syntax and options is given in Appendix G. ■

### *Contents*

General characteristics	5
Overview of the assembly process	6
Assembly files	7
Programming for the Macintosh	8
Macintosh libraries	9



---

## General characteristics

The MPW Assembler is a Macintosh program that reads your **source text** and creates a file of linkable **MC68xxx object code**. It has the following principal features, which help you build powerful assembly-language programs:

- It supports all the instructions and addressing modes for the MC68000, MC68010, MC68020, and MC68030 microprocessors, the MC68851 Paged Memory Management Unit (PMMU), and the MC68881 and MC68882 Floating-Point Coprocessors, in all usable combinations.
- It has powerful macro capabilities, which handle both positional and keyword macros. These capabilities resemble those of the macro facilities in the IBM 360/370 Assemblers.
- Its macro capabilities accept global and local variables (called SET variables) that allow macros to communicate with one another. SET variables may contain numbers, characters, strings, or arrays. You can use them with conditional and looping statements to control the generation of complex structures of object code.
- It gives you the choice of creating either a single object module or a series of separate object modules.
- It gives you full control over the generation of both code and data modules, including A5-relative data. You can share global data between your assembly-language routines and routines written in MPW Pascal and MPW C.
- It lets you specify the scope of all code and data definitions. You can make the objects they define accessible only within modules, within files, or between files.
- It lets you define templates that determine the mapping of data in memory. Their function is similar to that of Pascal records or C structures. You can use templates as data types in much the same way as you use record types in Pascal.
- It lets you store its global symbol tables in files and then use these files for new assemblies. This increases assembly speed and saves disk space.
- It can generate Pascal-formatted and C-formatted strings, as well as fixed-length strings.

---

## Overview of the assembly process

The MPW Assembler processes your source text in two **passes**. The first pass reads the source text, defines and expands all macros, and defines all symbols. It determines the length of the object code and resolves forward references. It also creates the following symbol tables:

- a **global symbol table** for symbols defined outside code or data modules
- a **macro symbol table** for all macro symbols and definitions
- a **local symbol table** for each code or data module

These operations do not create any object code.

As it starts to read each code or data module on its first pass, the Assembler creates an internal file in Macintosh memory containing a translation of your source text into **postfix notation**. At the end of each module, the Assembler performs its second pass, converting the postfix file into object code. The Assembler appends this code to the growing object code file. After processing each module, it releases memory held for the internal postfix file and the local symbol table.

Thus, the Assembler translates each module separately, releasing the memory used before the next module is started. Nevertheless, symbol tables, macro definitions, and the postfix file all compete for the Macintosh's memory. To permit you to assemble large programs with limited RAM space, the MPW Assembler lets the postfix file spill over onto a disk. When this happens, the Assembler returns a warning message at the end of the assembly process and assembly time increases by about 25 percent. Hence once there is enough RAM space available for its basic operations (including maintaining all symbol tables), the only memory limitation on the assembly process is the availability of disk space.

If you are generating an assembly listing, the Assembler creates a **scratch file** on the disk, in addition to the listing file. During the first pass, the Assembler writes source text lines that occur outside modules to the listing file directly. The Assembler writes lines that occur inside modules (including conditionally assembled lines and macro expansions) to the scratch file during the first pass, and then from the scratch file to the listing file during the second pass. To ensure that they appear in the correct location in the listing, the Assembler generates additional postfix code. As a consequence, assemblies that create listing files generate more postfix code.

During assembly, the Assembler sends errors and warnings to the **diagnostic output file** (the active window, unless you specify otherwise). If you use the **-p** Assembler option, described in Appendix G, the Assembler also writes progress and summary information to the diagnostic output. Status codes that the Assembler may send to the MPW Shell are listed in Table 1-1.

■ **Table 1-1** Assembler status codes

---

Code	Status
0	No errors detected in any files assembled
1	Assembler command line parameter or option errors detected
2	Assembly processing errors detected
3	Assembly terminated before completion

---

## Assembly files

By convention, you add the suffix **.a** to your assembly-language source text files. Object code files created by the Assembler are normally named after your source text file with the suffix **.o** added. However, you can change the name that the Assembler assigns to your object file by using the **-o** Assembler option, described in Appendix G. If you tell the Assembler to create a listing file, it will be named after your source text file with the suffix **.lst** added.

For example, given an assembly-language source text file **Name.a**, the Assembler will create an object code file **Name.a.o** and a listing file **Name.a.lst** from it.

In addition to the Assembler itself, the MPW disks contain library files of useful routines, together with the corresponding files of assembly-language interface statements, macros, and equates that access them. Your program can use any of these files. Files whose names end in **.a** contain assembly-language statements that you can include in your source text. Files whose names end in **.o** must be linked to your assembly. They contain executable code called by the assembly-language files. Most of the available libraries and their interface files are described in the *Macintosh Programmer's Workshop 3.0 Reference*; some are described in this book, in Appendix J.

---

## Programming for the Macintosh

There are four kinds of programs you can write for execution on the Macintosh:

- applications
- tools—programs that run under the MPW Shell
- desk accessories and other drivers
- 'CODE' resources such as cdevs and INTTs, which are used to customize the Macintosh environment, and XCMDs for extending the HyperTalk language used in HyperCard. (See the Macintosh Technical Notes for further information.)

General information about building and installing these kinds of programs is given in the *Macintosh Programmer's Workshop 3.0 Reference*. The following notes are specifically applicable to assembly-language programs:

- If an application contains one or more data modules containing DC or DCB directives, it must be linked with the library Runtime.o, which contains the data initialization routine `_DataInit`. Its first executable statement must be a call (JSR) to the entry point `_DataInit`. This entry point must also be declared as `IMPORT`. After returning from `_DataInit`, your program may unload the segment `%A5Init` that contains it, by calling the Macintosh routine `UnloadSeg`.
- Routines you can use with tools that run under the MPW Integrated Environment are described in Chapter 12 and Appendix F of the *Macintosh Programmer's Workshop 3.0 Reference*.
- Assembly-language desk accessories may not declare any global data. They must be linked with the file `DRVRRuntime.o`, which contains the main code module for all desk accessories. Use `Create Build Commands...` from the Build menu in MPW to help create build files for desk accessories and other types of 'CODE' resource.



---

## Macintosh libraries

*Inside Macintosh* describes an extensive group of **Macintosh library routines**, also called **operating-system routines** and **toolbox routines**. They perform jobs such as creating menus, windows, and dialog boxes, providing simple text editing, and accessing files and devices.

Many of the Macintosh library routines are implemented in the Macintosh ROM. You can call them from an assembly-language program by using machine instructions whose high-order four bits are %1010 (that is, whose opcodes begin with \$A). Such machine instructions are **unimplemented**, and using one of them invokes what is called an **A-trap**. The Macintosh **trap dispatcher** determines which of the library routines to call by examining the rest of the opcode.

The opcodes for various Macintosh library routines are defined by `OPWORD` directives contained in assembly-language files in the MPW folder {AIncludes}. If you include the appropriate files in your assembly, you can call the routines they cover by writing the routine identifiers (such as `_Read` instructions).

Certain Macintosh library routines are in library object files, instead of in ROM. They are flagged in *Inside Macintosh* with the notation “[Not in ROM].” You call these routines with `JSR` instructions. If you use any of them in your program, you must link your assembly with the MPW file {Libraries}Interface.o, which contains their code.

Additional information about calling Macintosh operating-system and toolbox routines from assembly-language programs is contained in the Using Assembly Language chapter of *Inside Macintosh*. The include files and library files supplied with MPW are described in the *Macintosh Programmer's Workshop 3.0 Reference*.



## Chapter 2 Coding Conventions

THIS CHAPTER DESCRIBES THE SYNTAX RULES AND OVERALL form required for source text that is to be processed by the MPW Assembler. ■

### *Contents*

Source text structure	13
Scope of definitions	15
Imported and exported objects	17
@-labels	17
Summary	18
Segmentation	18
Machine instruction syntax	19
The label field	20
The operation field	21
The operand field	23
Comments	24
Symbols	25
Identifiers	25
Numeric constants	26
Strings	27
Expressions	28
Evaluation of expressions	30
Absolute and relocatable expressions	31
Absolute expressions	32
Relocatable expressions	33



---

## Source text structure

The source text formats for most higher-level programming languages are similar in structure. They consist of related procedures and functions plus various forms of data. The programs that interpret them differ mainly in the ways they support relationships among the routines and data. The MPW assembly language, although not a higher-level language, includes many of the programming facilities found in higher-level languages.

In order to understand the structure of MPW assembly-language source text, it is helpful to understand the various components that make up a linked and executable program. Therefore, this discussion begins by defining some terms that describe an executable Macintosh program, together with how they relate to an assembly source text and the environment in which the program is executed.

Each line of MPW Assembler source text is either a **machine instruction statement** or a **directive statement**. Machine instruction statements generate executable code, using MC68xxx instructions. Directive statements are commands to the Assembler to perform certain operations during assembly. The syntax rules for writing machine instruction statements are given later in this chapter. The syntax rules for writing directive statements are given in "Directive Formats" at the beginning of Chapter 4.

Every executable assembly-language program is built from a collection of **object files**. Each object file corresponds to one assembly and is made up of a collection of code and data **modules**. Each module contains one contiguous piece of code or data. Data modules represent static data, because the data space is defined before the program begins and the data remains accessible during the entire execution of the program.

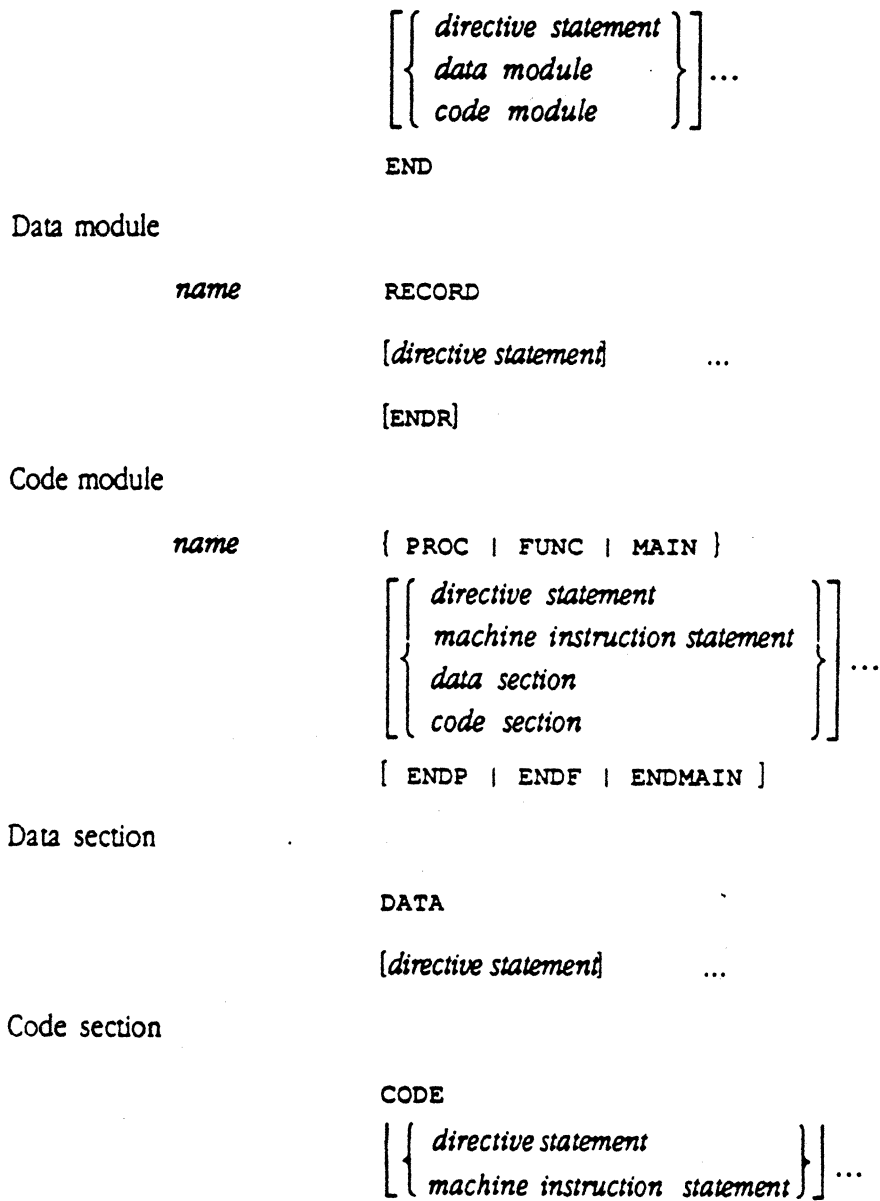
When you link a program, the Linker groups all the code modules together and makes a separate grouping of all the data modules. Thus a linked object file consists of two parts: a collection of code modules and a collection of data modules.

In the Macintosh, the Segment Loader takes the collection of data modules and loads them into an area called the **application globals area**. This area is just below the area pointed to by register A5, called the **application parameter area**. Thus when the Linker adjusts code references to data in the data modules, it does so by setting negative offsets relative to A5, the assumed base register for data access.

References by code to other code are made by jumping indirectly through a structure called the **jump table**, which is also built by the Linker and loaded by the Segment Loader just above the application parameter area. So the jump table is accessed by positive offsets from A5. A map of all these memory areas is included in Figure 9 of the Memory Manager chapter of *Inside Macintosh*.

Figure 2-1 is a syntax diagram that covers the overall structure of an assembly-language source file. The modules and directive statements in such a source file may occur in any order, subject only to the scope rules given later in "Scope of Definitions."

■ **Figure 2-1** Source text structure



Each module starts with a **module directive** (`PROC`, `FUNC`, `MAIN`, or `RECORD`). Each module terminates at the start of the next module directive, at the matching `ENDx` (`ENDP`, `ENDF`, `ENDMAIN`, or `ENDR`), or at the end of the source text. The end of the source text is indicated by an `END` directive.

Code modules are always introduced explicitly by either a `PROC`, `FUNC`, or `MAIN` directive. There is no structural difference between `PROC` and `FUNC` directives; `FUNC` is used instead of `PROC` only for documentation purposes. `MAIN` is essentially the same as `PROC` or `FUNC`, but has the additional function of indicating that this module is the **main code module**, and that its first instruction is the execution starting point for the program. There must be exactly one main code module in a linked program.

There are three ways to declare data in an assembly-language source file:

- As a data module introduced by the `RECORD` directive. Such a module generally contains data-definition and storage-allocation statements. It may also contain initialized values. All the data defined between the `RECORD` directive and its matching `ENDR` (or the start of the next module) generates a single data module.
- As a data module corresponding to one data-definition statement. Before the first module, or between explicitly declared modules, you may write directive statements. Such statements outside of explicitly declared modules define their own data modules, one corresponding to each statement.
- As data that is part of a code module. Although you use the `PROC`, `FUNC`, and `MAIN` directives to indicate the start of a code module, you may generate an associated data module inside the code module, using the `CODE` and `DATA` directive. `CODE` and `DATA` may be used only inside a code module (`PROC`, `FUNC`, or `MAIN`); they indicate a switch from code to data (`DATA`), and then back to code again (`CODE`). Hence they delimit sections of code or data within the code module. Code in the code sections is generated contiguously—the first byte of one code section immediately follows the last byte in the previous code section. Similarly, the data in the data sections is contiguous.

---

## Scope of definitions

The **scope** of a definition is the area of source text in which the code or data object it defines is accessible—that is, the area in which the object can be accessed by code or data statements. Scope rules permit you to restrict the scope of definitions. This lets you allow communication among the various routines of your program, while at the same time making selected objects inaccessible to other routines. Selectivity of scope promotes structured programming and helps you avoid identifier conflicts.

Here are the scope rules:

- All code or data definitions in a source file have either **global** or **local** scope.
- Local definitions override global definitions.
- The scope of a global definition extends from the point at which it occurs in the source file to the end of that file. Global definitions include those declared outside of a code or data module as well as definitions of code and data module identifiers. All identifiers assigned to global objects must be unique within the assembly.
- All code or data labels must be declared or defined before they are used. In order to access a label prior to its definition in the file, you must declare it with an **IMPORT** or **EXPORT** directive before the access.
- The Assembler permits field identifiers within a data module (created by the directive **RECORD**) to be accessed as **qualified** identifiers. Qualified identifiers are written in the form *modname .fieldname*, where *modname* is the data module identifier and *fieldname* is a data-definition field identifier, as defined within the data module. Field identifiers accessed in this way have global scope.
- A definition is considered to have local scope if it occurs inside a code or data module. Local objects may be accessed only from within the module; you may use the object's identifier in different modules or outside the module without causing an identifier conflict.
- The global/local scope rules may be overridden with the **ENTRY**, **EXPORT**, and **IMPORT** directives described in this chapter and in Chapter 4.
- **ENTRY** forces specific identifiers to be global. An identifier that is to be declared as **ENTRY** must be so declared before it is defined. From that point on, the identifier follows the same rules as global identifiers. This means that **ENTRY** may be used to access identifiers defined later in your source text, such as labels in subsequent **PROC** directives.
- Since all data objects outside of modules, as well as the module identifiers themselves, have global scope, they are implicitly declared as **ENTRY** by the Assembler. For documentation purposes, a module identifier may also be declared explicitly as **ENTRY**.



## Imported and exported objects

Local or global code or data objects may be made accessible to source text files other than the file in which they are defined. Objects defined in a file, intended to be accessed outside it, are said to be **exported**. Objects accessed from outside the file in which they were defined are said to be **imported**. Thus an exported object in an object file can be imported into any number of other object files. You export and import objects by using the `EXPORT` and `IMPORT` directives.

Using `EXPORT` inside a code or data module declares specified local identifiers as exported. You must use `EXPORT` before defining the specified identifiers. The identifiers may then be accessed from other files that import those identifiers. Since an exported local identifier is made accessible outside of the module in which it is defined, `EXPORT` promotes local identifiers to global scope within the same source text file, just as `ENTRY` does. Module identifiers, code or data labels, global data definitions, and storage-allocation identifiers may be exported.

Once an object's identifier is declared as `EXPORT`, other source text files may import the identifier by using `IMPORT`. The `IMPORT` directive declares specified identifiers as local or global, depending on where the `IMPORT` statement is used within the file. If `IMPORT` is used inside a module, then the identifiers are local to that module. Using `IMPORT` outside a module declares the identifiers as global to the rest of the file.

Since `EXPORT` identifiers are global to the file in which they are declared, the Assembler treats references to such identifiers from modules other than the one that actually defines the identifier as imports of those identifiers. For documentation purposes, however, you can always import such identifiers explicitly by using `IMPORT` in the same file.

## @-labels

Label identifiers that begin with an at symbol (`@`) are called **@-labels**. They have more limited scope than other labels and can't be used in directives or outside modules. Specifically, the scope of an @-label extends through the source text, in both directions, to the nearest label that doesn't begin with `@`. You may redefine an @-label, but not in the scope of another instance of the same @-label. All @-labels defined or used inside macros follow the same rules, but in addition their scope is limited to the body of the macro. Any @-labels passed as macro parameters retain the scope they had when the macro was called, with certain restrictions. For further information about passing @-labels to macros, see "Operand Syntax" in Chapter 5.

## Summary

Here is a summary of the kinds of identifiers used in definitions of different scope.

- **Temporary scope (can be accessed only within a part of a module)**
  - @-labels (beginning with @)
- **Local scope (can be accessed only from within the module; overrides global declarations)**
  - All identifiers defined within a code or data module
  - Identifiers imported by using `IMPORT` within a module
- **Global scope (can be accessed from the point of definition to the end of the file)**
  - Code and data module identifiers
  - All identifiers defined outside of code and data modules
  - Identifiers imported by using `IMPORT` outside of any module
  - Identifiers declared as `ENTRY`
  - Qualified data module identifiers
- **Identifiers accessible between files**
  - Local identifiers declared as `EXPORT` (`EXPORT` used inside a module)
  - Global identifiers declared as `EXPORT` (`EXPORT` used outside any module)

---

## Segmentation

In addition to dividing your program into code modules, you can associate groups of one or more code modules into **segments**. As your program is executed, the Macintosh Segment Loader will load all the modules in each segment at the same time, whenever any one module in the segment is called. This lets you use the same memory space for different modules as long as they are in different segments. For example, you may have a collection of modules needed only for initialization of your program. These modules could be in one segment and the rest of your program in another segment. During initialization, only the initialization segment need be loaded. After initialization, that segment can be unloaded (by a call to `UnloadSeg`) and the same memory space reused by the remainder of your program. Segments and the Segment Loader are further discussed in the Segment Loader chapter of *Inside Macintosh*. Data initialization is discussed in the *Macintosh Programmer's Workshop 3.0 Reference*.

The `SEG` directive, described in Chapter 4, lets you group a code module or a collection of code modules into a particular segment. Only code modules may be placed in segments; data modules are not affected by `SEG` directives.

Each `SEG` directive specifies a name for the succeeding segment. All code modules up to the next `SEG` directive are grouped in the specified segment, beginning at the next code-module directive.

Code modules grouped in the same segment do not have to be contiguous in the source file. Code modules belonging to different segments may be mixed in your source text as long as they are covered by the appropriate `SEG` directives. The `SEG` directive is further discussed in "Linker and Scope Controls" in Chapter 4.

◆ *Note:* Segment names are case-sensitive. Be careful to capitalize them consistently.

---

## Machine instruction syntax

Machine instruction statements are written in four **fields**—the **label field**, the **operation field**, the **operand field**, and the **comment field**. These fields must be separated by one or more spaces (ASCII code \$20) or tabs (ASCII code \$09), and must be written in the order given. Total statement size is limited to 255 characters. You may continue writing a statement on the next line if you follow these rules:

- The fields must remain in their proper sequence: label, operation, operand, and comment.
- The fields must be separated by one or more spaces or tabs.
- Only the operand and comment fields may be continued. The label and operation fields must be completed in the first line of the statement, including at least one space following the operation entry.

Each continued line (after the first line) starts at the first character on that line that is not a space or tab; leading spaces or tabs on continued lines are ignored. For further information about continuing machine instruction statements, see "The Operand Field" later in this chapter.

---

## The label field

The **label field** is the first field in a source text line. It may be empty or it may contain an identifier. The syntax rules for identifiers are given in "Identifiers" later in this chapter.

If the label field contains an identifier, it need not begin in the first character position on the line. However, if it contains an identifier that begins after the first character position (that is, if the identifier is preceded by one or more spaces or tabs), the label field must be terminated by a colon. Otherwise the label field may be terminated by either a colon, a tab, or one or more spaces. The colon, if used, is not part of the identifier.

Within code and data modules, the label field may be the sole field of a source text line, in which case it terminates with the return character that ends the line. In code modules, the Assembler always aligns such label positions to start on a word boundary. When a source text line contains only the label field and the comment field, they must be separated by a semicolon (;) preceded by at least one space character.

Here are some examples of valid label syntax:

```
label      MOVE.W   D0,D1
 label:    MOVE.W   D0,D1
label
label      ;Comment
```

The first line shows a label that begins in the first character position, and hence can be terminated by tabs or spaces. The second line shows a label preceded by a space; it must be terminated by a colon. The third line contains only a label. The last line contains a label and a comment, which must be separated by a semicolon preceded by at least one space or tab.

All labels that begin with an at symbol (@) are called @-labels. They can be used only inside modules, as described in "Scope of Definitions," given earlier in this chapter.

The Assembler allows labels for all instructions, macro calls, and directives that define data structures or values. For instructions and data-definition directives, the label is given a value equal to the location-counter value associated with the first byte of the instruction or data. For macro calls and other directives, the label's value is defined as a function of the macro call or directive.

---

## The operation field

The operation field contains the mnemonic operation code specifying the desired machine instruction or Assembler directive. Mnemonic operation codes conform to the rules for identifiers given later in this chapter. Valid operation codes include the following:

- mnemonics for the MC68000 and MC68010 instructions described in the Motorola *MC68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual*
  - mnemonics for the MC68020 instructions described in the Motorola *MC68020 32-Bit Microprocessor User's Manual*
  - mnemonics for the MC68030 instructions described in the Motorola *MC68030 Enhanced 32-Bit Microprocessor User's Manual*
  - mnemonics for the MC68851 PMMU coprocessor instructions described in the Motorola *MC68851 Paged Memory Management Unit User's Manual*
  - mnemonics for the MC68881 and MC68882 floating-point coprocessor instructions described in the Motorola *MC68881/MC68882 Floating-Point Coprocessor User's Manual*
  - the Assembler directives, including macro instructions, described in this book
- ◆ *Note:* Some mnemonics have been changed to eliminate ambiguities and to conform to the Motorola assembler forms. If in doubt, check your mnemonics with those listed in Chapter 3 and Appendix F.

The operation field must be preceded by at least one space or tab. The Assembler ignores uppercase and lowercase distinctions when reading it.

Certain machine instruction mnemonics include condition codes, indicated by the symbol *cc*. A list of the condition codes the MPW Assembler accepts is included in Appendix F.

Many instructions and directives can operate on more than one data size. For these operations, the data size must be specified as part of the mnemonic; otherwise a default size is assumed. The size is specified by appending to the mnemonic a period (.) followed by one of the qualifier letters shown in Table 2-1.

■ **Table 2-1** Data size qualifiers

Letter	Name	Data Size
B	Byte	8 bits
W	Word	16 bits
L	Long word	32 bits for data; signed offset for branch instructions
S	Short	8-bit signed offset, -128..127, for branch instructions
D	Double long word	64 bits; for certain MC68851 and MC68030 registers only
S	Single precision	32-bit IEEE format for binary reals: 8 exponent bits, 23 mantissa bits, 1 sign bit; MC68881 and MC68882 only
D	Double precision	64-bit IEEE format for binary reals: 11 exponent bits, 52 mantissa bits, 1 sign bit; MC68881/MC68882 only
X	Extended	96-bit IEEE format for binary reals: 15 exponent bits, 64 mantissa bits, 1 sign bit, 16 reserved bits; MC68881/MC68882 only
P	Packed BCD	96-bit packed BCD format for real strings: 3 decimal digits exponent, 17 decimal digits mantissa, 4 bits sign and range, 12 reserved bits; MC68881/MC68882 only

In macro calls, the period may be followed by any sequence of characters, as long as none of them are spaces or tabs. The meaning of such a qualifier is a function of the macro definition associated with the call. See "Defining Macros" in Chapter 5 for further details.

Ordinarily, the default data size qualifier is word (W) for MC680X0 and MC68851 instructions and extended (X) for MC68881/MC68882 instructions. Some instructions do not permit a data size specification, since the size is implicit in their operation.

In some cases, the Assembler accepts a **generic form** for an instruction and assembles a more appropriate form. The instruction ADD, for example, is translated into ADD, ADDA, ADDQ, or ADDI, depending on context. The generic instruction formats are listed in Appendix A. The reasons for using them fall into three overlapping categories:

■ **Optimization**

- Instructions can often be encoded into a more compact (and generally faster-executing) form that is not the same as the original instruction. An example of such an instruction is `SUBA A $n$ , A $n$`  in place of `MOVE #0, A $n$` . When an instruction is optimized, the object code generated is different, but the mnemonics are not changed in the listing.

- **Convenience**

- Instructions may need to be encoded based on context, such as an `ADDI` in place of an `ADD`. Also, alternate mnemonics may make coding easier and more readable. For example, `BZ` (branch if zero) would replace `BEQ`.

- **Compatibility**

- Instructions may need to be translated for compatibility with the Lisa™ Workshop Assembler (TLA), the Macintosh 68000 Development System Assembler (MDS), or the Motorola assembler. Examples here include `BHS` (branch on high or same) for `BCC` (branch on carry clear), and `BLO` (branch on low) for `BCS` (branch on carry set).

You can control whether the Assembler optimizes instructions by using the `OPT` directive, described under "Assembly Options" in Chapter 4.

---

## The operand field

Many instructions and directives require operands as part of their specification. The operand field follows the operation field and must be separated from it by at least one space or tab. The operand field may be empty, or it may be composed of one or more subfields separated by commas.

The `BLANKS` Assembler directive controls where tabs or spaces may be placed within the operand field. With `BLANKS OFF`, they may occur only after commas separating operand subfields and between paired parentheses, brackets, or braces. With `BLANKS ON`, tabs and spaces may be placed anywhere in the operand field except within symbols. (Symbols are discussed under "Symbols" later in this chapter). With `BLANKS ON` (the preset condition), a semicolon is always required to separate the operand field from the comment field. For further information, see the discussion of `BLANKS` under "Assembly Options" in Chapter 4.

If you intend to continue an operand field on the next line, you must place the **backslash continuation character** (`\`) in the operand field before any semicolon that precedes a comment. The backslash character may not be used to continue a single symbol. This means that line continuation can occur only between symbols. Furthermore, with `BLANKS OFF` you must place the continuation character so that the Assembler treats it as part of the operand field—that is, immediately before or after a symbol, or among tabs or spaces that the Assembler will ignore because the operand field is not yet complete.

Here is an example of the correct way to continue an operand field:

```
EXPORT  name1, name2, name3, \  
        name4, name5:DATA           ; Looks good
```

Here is an example of an incorrect operand continuation:

```
EXPORT  name1, name2, na\  
me3:DATA ; Broken symbol
```

---

## Comments

You can insert a comment into your source text in two ways: as a comment field or as a **comment line**. Comments are ignored by the Assembler and may contain any characters except return (ASCII \$0D). Comments are intended for your use in documenting your program.

The comment field is the last field in a source text line; it must be separated from the preceding field by at least one tab or space. As mentioned earlier in this chapter in "The Operand Field," the setting of the `BLANKS` directive influences whether or not comments must be preceded by semicolons. In statements where an optional opcode, operand field, or subfield is omitted but a comment is desired, the comment must always be separated from the rest of the line by a semicolon preceded by one or more tabs or spaces, even with `BLANKS OFF`.

An entire line may be used for a comment by placing an asterisk (\*) or semicolon (;) in the first character position of the line. On lines which contain only a label, the semicolon convention must be used, even with `BLANKS OFF`.

To continue a comment that began with an asterisk (\*), enter a backslash continuation character (\) and go immediately to the next line. Comments that begin with semicolon (;) cannot be continued.

The Assembler ignores lines that contain no characters. They are treated like comment lines and can be used to separate sections of code or comments.

Here are some examples of valid comment syntax:

```
label1  MOVE.W    D0,D1      This is a comment with BLANKS OFF  
label1  MOVE.W    D0,D1      ; This is a comment with BLANKS ON  
label2                                     ; No opcode-- semicolon required  
label3  PROC                                     ; Semicolon required because  
                                           ; PROC has optional parameters
```

- \* This is a whole-line comment.
- ; This is also a whole-line comment.
- \* This is a comment that is too long to fit entirely on one line \  
and therefore is continued on a second line.
- \* However, you can also continue a comment on a second line without using the  
continuation character, by starting the second line with another asterisk.



---

## Symbols

Except for comments, all fields of an Assembler statement are composed of **symbols**. A symbol is a character or a combination of characters used to represent an identifier, a numeric constant, or a character string.

Different kinds of symbols are allowed in the different fields of assembly-language source text:

- The label field may contain only a single identifier.
- The operation field must contain a single MC68xxx instruction mnemonic, a macro call, a directive name, or an identifier defined by `OPWORD`.
- The operand field may contain one or more symbols or expressions composed of symbols of any kind.

The following sections discuss the symbols accepted by the MPW Assembler. "Expressions," later in this chapter, discusses the expressions that you can form out of symbols.

---

## Identifiers

Names and labels are **identifiers**. The first character of an identifier must be an uppercase or lowercase letter (A..Z, a..z), an underscore ( \_ ), or an at symbol ( @ ). The Assembler treats any label that begins with @ as an @-label.

Subsequent identifier characters can be letters, digits (0..9), underscores ( \_ ), dollar signs ( \$ ), number signs ( # ), percent signs ( % ), or at symbols ( @ ). An identifier can be any length, but only the first 63 characters are significant. By using the `CASE` directive, you can specify whether uppercase and lowercase letters are to be treated as different or the same. See "Assembly Options" in Chapter 4 for more information.

Some examples of valid identifiers are shown here:

```
BYTE      NextChar      ApplZone      _trap      X      @2      A_1      A#2
Start     Next_Char     inverseBit   Num#65     a#      _      AS2      A@2
```

A special identifier symbol is used to refer to the current value of the **location counter** in a module or template. This symbol is the asterisk ( \* ). It may appear only in the operand field. It stands for the address of the first byte of currently available storage after any required boundary alignment. Using the asterisk in the operand field of a statement is the same as placing a label in the label field of the statement and then using that label in the operand field of the same statement.

The Assembler uses the location counter referred to by the asterisk symbol to assign code and data module addresses to statements. It is the Assembler's equivalent to a computer's instruction counter. Since all modules are relocatable, all modules are assembled with their addresses relative to zero. Therefore the location counter is a zero-relative offset to the address of the start of the current module. Since it is an offset, the location counter may also be used in templates. Hence each module and template may be viewed as having its own location counter.

---

## Numeric constants

You can express **numeric constants** in your source text in either **decimal**, **hexadecimal**, **binary**, or **floating-point** form.

These are the syntax rules for expressing numeric constants.

- A decimal number is formed as a string of decimal digits (0..9), as shown here:

```
123
5
32
```

- A hexadecimal number is specified by a dollar sign (\$) followed by a sequence of hexadecimal digits (0..9, A..F, or a..f), as shown here:

```
$123
$1A3C
$FFFF
$a1c2
```

- A binary number is specified by a percent sign (%) symbol followed by a sequence of binary digits (0 or 1), as shown here:

```
%1010
%101
%1011101
```

- A floating-point number is specified by enclosing a decimal or hexadecimal number in quotation marks ("), ASCII \$22). Decimal numbers, for this purpose, may include any of the forms listed in the *Apple Numerics Manual, Second Edition*, Table 3-2. A hexadecimal floating-point number must begin with a dollar sign, following the format given above. Here are some examples:

```
"123"      "123.4E-12"  "123."      ".456"      "nan"
"-0"      "-INF"       "NAN(12)"   "-Nan()"    "$3F800000"
```

The MPW Assembler interprets decimal, hexadecimal, and binary constants as signed 32-bit values. For example, \$FFFF is interpreted as the value 65535, not -1. If you want -1, you must write it in decimal as -1 or in hexadecimal as \$FFFFFFFF. The Assembler interprets floating-point numbers as required by the MC68881/MC68882 instruction that uses them—as single, double, extended, or packed BCD.

- ◆ *Note:* The Assembler pads incomplete hexadecimal floating-point numbers with zeros at the *right* end. For example, if you write \$A123 as an operand for an MC68881/MC68882 instruction that requires eight-byte data, the Assembler will interpret it as the number \$A1 23 00 00 00 00 00. (Spaces added for clarity.)

---

## Strings

A **string** is a sequence of one or more ASCII characters (including spaces and tabs) enclosed in single quotation marks ( ' , ASCII \$27). Within a string, two single quotation marks in succession represent one single quotation mark. Some examples of strings are

```
'Hello'
'don't'
''''      (Generates one single quote)
```

There are restrictions on how long a string can be, as well as how it is interpreted by the Assembler. These restrictions depend on its context and form, as explained here.

A **string constant** used to represent an integer value in an arithmetic expression is limited to four characters. The Assembler evaluates each character as having the value of its ASCII code. It treats such a string as a right-justified 32-bit value, padded on the left with zeros. Here are some examples:

SUB	#'a'-'A',D0	Constant represents the value 32
SUB	#'a'-\$41,D0	Same as the previous example
MOVE.B	#'1',D0	Put \$31 into low byte of D0
MOVE.W	#'1',D0	Put \$0031 into low word of D0
MOVE.W	#'12',D0	Put \$3132 into low word of D0
MOVE.L	#'1',D0	Put \$00000031 into D0
MOVE.L	#'12',D0	Put \$00003132 into D0
MOVE.L	#'123',D0	Put \$00313233 into D0
MOVE.L	#'1234',D0	Put \$31323334 into D0
MOVE.L	#'12'+1,D0	Put \$00003133 into D0

Strings used under any of the following conditions may be of any length up to the line-length limit of 255 characters:

- Strings defined as data operands to DC and DCB directives.
- Strings used in relational expressions.
- Strings used to assign values to macro variables.
- Strings used as source operands for PEA and LEA instructions. This is the only case where an arbitrary-length (within the 255 character line-length limit) string may be used in a machine instruction. It represents an instance of a literal. Literals are discussed in Chapter 3 under "Special Address Formats."

Using the `STRING` directive, arbitrary-length string constants may be generated in any of three formats, depending on the option specified:

- **As-is string:** the string is generated as specified.
- **C string:** the string is generated with a zero-value byte following its last character. This is the string format used by C.
- **Pascal string:** the string is preceded by a length byte. This, the default setting, is the format of strings used by Pascal and the Macintosh library routines. Pascal strings are limited to 255 characters.

For further information about the `STRING` directive, see "Assembly Options" in Chapter 4.

If a string variable appears as the value of a macro parameter, the Assembler interprets it as a string when it appears in a relation and as an integer when it appears in an arithmetic expression. For example, suppose the string '123' is the value of the macro parameter `&i`. When used in the expression `&i = '123'`, it would appear as a string. In the expression `&i + 10`, it would yield the 32-bit value of the integer 133. This is different from the case of a string constant '123', which is treated as the value `$00313233`.

Declared macro string variables are always treated as strings and may be used only as strings. Macros have typed variables, as described in Chapter 6. Such variables declared as specific types may be used only in contexts appropriate to their type.

---

## Expressions

Expressions are used either in the operand field of source text or as SET array variable subscripts (defined in Chapter 6 under "SET Array Variables"). They may be composed of a single term or a combination of terms, with each term being either an identifier, a constant, the location-counter symbol (\*), or a macro function call. Integer terms are treated as 32-bit signed values, and are combined by arithmetic, logical, shift, and relational operators. Macro string terms may be combined only with relational operators.

The MPW Assembler recognizes the operators shown in Table 2-2. They are listed from highest precedence to lowest. Groupings indicate operators of the same precedence.

■ Table 2-2 Operators

Precedence	Symbols	Operation
Highest	( )	Grouping by parentheses
	~	Ones complement
	¬ NOT	Logical not
	-	Unary negation
	*	Multiplication
	/ DIV +	Division
	// MOD	Modulus division
	+	Addition
	-	Subtraction
	>>	Shift right
	<<	Shift left
	=	Equal to
	<> *	Not equal to
	<	Less than
	>	Greater than
	<= ≤	Less than or equal to
	>= ≥	Greater than or equal to
	** AND	Logical and
	++ OR	Logical or
Lowest	-- XOR	Logical exclusive-or

The rules for writing expressions are as follows:

- Only parentheses and the +, -, ~, ¬, and NOT operators are allowed at the start of an expression.
- Subexpressions are designated by enclosing the subexpression in parentheses.
- An expression may not contain two terms or two operators (other than parentheses) in succession.
- Parentheses may be nested to a maximum depth of 9 pairs.
- Arithmetic expressions should contain a maximum of 20 terms.
- If an expression is enclosed in parentheses, the Assembler ignores blanks within the expression regardless of the setting of the BLANKS directive.

- The multicharacter operators `DIV`, `MOD`, `AND`, `OR`, `XOR`, and `NOT` must be separated from identifiers by at least one space. Hence these operators may be combined with identifiers only if the `BLANKS` directive is `ON` (the preset mode), or if the expression containing them is enclosed in parentheses.
- Floating-point constants may be enclosed in parentheses but may not be combined with any of the other operators in Table 2-2.

---

## Evaluation of expressions

A single symbol is a single-term expression with the value represented by the symbol. The Assembler reduces multiterm expressions to single values, following these rules:

- Each numeric term is given a 32-bit value. Overflows are ignored.
- Operations are performed from left to right, following the precedence indicated in the operator table above.
- A parenthesized subexpression is reduced to a single value. The resulting value is then used in computing the final value of the expression.
- When parenthesized subexpressions are nested, the innermost subexpression is evaluated first.
- Every expression is computed as a 32-bit signed value. The limits on the final value depend on how the expression is used.
- Division always yields an integer result; any fractional portion of the result is dropped.
- Division by 0 yields 0 as the result.
- The relational operators assign the absolute value 1 when the relation is `true`, and the absolute value 0 when the relation is `false`. The comparison is algebraic, except when two character strings are compared. See "Boolean Control Expressions" in Chapter 7 for a discussion of the rules governing string comparisons.
- The `NOT` operator is equivalent to an exclusive-or with 1—that is, `¬e` is equivalent to the expression `e XOR 1`. This lets you negate Boolean expressions containing relational operators.
- The shifting operators `<<` and `>>` shift the left operand by the number of bits specified in the right operand. Zeros are shifted into vacated bit positions. Bits shifted out are lost. Shifting by more than 32 bits does not generate an error.

Expressions used as operands for DC and DCB directives or as literal operands for PEA and LEA statements may have either string or integer values. The Assembler decides which type the expression has by checking its first symbol. If the first symbol can be interpreted as a string, the Assembler assumes the whole expression is a string. Thus the Assembler may fail to evaluate certain ambiguous expressions, such as 'a' - 32, as you may expect, because the first symbol is a string constant. To force the Assembler to evaluate an expression arithmetically, enclose it in parentheses; for example, ('a' - 32).

Here are some examples of valid expressions:

*	A * 10	(a AND b)
* + 100	Alpha + (i > j) * 10	(a ** b) ++ (c ** d)
Rec.Field+10	-64	(a AND b) OR (c AND d)
Alpha - Beta	(a - b) + (c - d)	~(x + 10)
(a - b) / (20 + (c - d))	NOT(a OR b)	a >> b
'a' - 32	'ab' + \$8000	10 + x.y

---

## Absolute and relocatable expressions

When an identifier is used as a label, the Assembler assigns it a value. This value is **absolute** or **relocatable**, depending on the kind of statement or directive being labeled. Absolute values are unaffected by their code module's location in memory and have the same values at assembly time as they do at run time. Relocatable values represent addresses.

Code and data module identifiers and code or data labels are relocatable. All code modules are relocatable and data modules are relocated relative to register A5. Template identifiers and fields are absolute.

Using the location-counter symbol (\*) in an Assembler statement is the same as placing a label in the label field of the statement and then using that label in the operand field of the same statement. Since using the location counter is equivalent to using the label, it may be considered either relocatable or absolute—relocatable when used in a code or data module, absolute when used in a template.

The use of absolute and relocatable values in expressions causes the expression and its resulting value to be either absolute or relocatable. The following sections describe how to create absolute and relocatable expressions.

## Absolute expressions

An absolute expression may be an absolute symbol representing an absolute value, or any arithmetic combination of absolute symbols. The resulting value is an absolute value. All operators are allowed in absolute expressions, subject to the rules given above under "Evaluation of Expressions."

An absolute expression may contain relocatable values, alone or in combination with absolute terms. All terms in such an expression must already have a value; there may be no forward references. If there are relocatable terms there must be exactly one pair of them, and the relocatable terms

- may be used only in effective addresses of machine instructions and DC or DCB directive statements
- must access the same segment
- must refer both to code or both to data
- must consist of terms with opposite signs (+ and -)

The pairing of relocatable terms of opposite sign is allowed in an absolute expression because the subexpression involving the difference between the relocatable terms cancels the effect of relocation, thus producing an absolute value.

The following examples illustrate absolute expressions. In these examples, *r1* and *r2* are relocatable symbols; *a1* and *a2* are absolute symbols.

```
a1
a1 + 100 - a2
a1 * a2
r1 - r2
(r1 - r2) + a1 * 10
r1 + a1 * 10 - r2
```



## Relocatable expressions

A relocatable expression may contain relocatable values, alone or in combination with absolute terms, provided that it conforms to these rules:

- There must be either one or three relocatable terms.
- If there are three relocatable terms, two of them must be paired, as described earlier in this chapter in "Absolute Expressions."
- Relocatable symbols may be combined only with the + and - operators.

A relocatable expression reduces to a single relocatable value. This value is derived from the odd relocatable term, adjusted by the values of the absolute terms.

In effective addresses, the Assembler assumes that all imported code or data symbols and all forward references to undefined symbols are relocatable.

The following examples illustrate relocatable expressions. In these examples, *r1* and *r2* are relocatable symbols; *a1* is an absolute symbol; and *i1* and *i2* are imported symbols.

```
r1 + 10  
r1 + (a1 * 10) - r2  
i1  
i1 + 10 - i2  
i1 + 10
```



## Chapter 3 Address Syntax

THIS CHAPTER COVERS THE SYNTAX RULES FOR writing MC68000, MC68010, MC68020, and MC68030 addresses in MPW assembly-language source text. ■

### *Contents*

Addressing modes	37
Ambiguities and optimizations	41
Forward-reference addressing	43
Registers	44
Special address formats	46
MC68xxx instructions	46
MOVEM: Multiple moves	46
MC68020 instructions	47
MULS and MULU: Signed and unsigned multiplication	47
DIVS and DIVU: Signed and unsigned division	47
TDIVS and TDIVU: Truncated signed and unsigned division	47
PACK and UNPK: Packing and unpacking	48
CAS and CAS2: Comparing and swapping	48
Bit field instructions	49
Tcc and TPcc: Trap on condition	49
Assembler control	49
The MC68030 processor	49
Assembler control	50
MC68020 statements you can use	50
MC68851 instructions you can use	50

MC68881 and MC68882 instructions 52  
    FMOVEM with explicit register lists 52  
    FMOVE with packed BCD data 52  
    FSINCOS: Simultaneous sine and cosine 53  
    FTcc and FTPcc: Floating-point trap on condition 53  
    FTEST: Text operand and set floating-point condition codes 53  
MC68851 instructions 53  
Literals 55

---

## Addressing modes

The MC68000 **effective addressing modes** are fully discussed in the Motorola *MC68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual*. Additional addressing modes are available for the MC68020/MC68030. Both the MC68000 and MC68020 addressing modes are discussed in the *MC68020 32-Bit Microprocessor User's Manual*. MC68030 extensions are discussed in the *MC68030 Enhanced 32-Bit Microprocessor User's Manual*. If you are not familiar with how the Motorola addressing modes work, you should read one of these books before trying to understand this chapter. The symbols used in MPW assembly-language addresses are listed in Table 3-1.

■ **Table 3-1** Address symbols

---

ID	Meaning
<i>An</i>	Address register <i>n</i> , where <i>n</i> is a number in the range 0..7
<i>Dn</i>	Data register <i>n</i> , where <i>n</i> is a number in the range 0..7
<i>Rn</i>	Register <i>n</i> , either address or data, where <i>n</i> is a number in the range 0..7
<i>Xn</i>	Index register <i>n</i> , where <i>n</i> is a number in the range 0..7. An index register may be a data register ( <i>Dn</i> ) or an address register ( <i>An</i> ), optionally followed by a period and a <i>w</i> or <i>L</i> size designation (16 or 32 bits, respectively).
<i>*s</i>	Scaling factor, where <i>s</i> is an absolute expression which must produce the value 1, 2, 4, or 8. Values 2, 4, and 8 can be used only with the MC68020 and MC68030. The default value for <i>s</i> is 1. If you omit <i>s</i> , you must omit the asterisk also.
<i>PC</i>	The program counter
<i>ae</i>	An absolute expression
<i>re</i>	A relocatable expression
<i>d</i>	An absolute ( <i>ae</i> ) or relocatable ( <i>re</i> ) expression resolving to 8 or 16 bits, depending on the addressing mode.
<i>bd</i>	Base displacement that is added before indirection occurs (MC68020 and MC68030 only). This is defined as an absolute expression for addressing mode 6 and a relocatable expression for addressing mode 73. A word or long word is generated for the <i>bd</i> as a function of its value, or for forward references as specified by the <code>FORWARD</code> directive, discussed in Chapter 4. You may, however, explicitly control the generated size by using the syntax ( <i>bd</i> ) <i>w</i> or ( <i>bd</i> ) <i>L</i> .

(continued)

■ **Table 3-1** (continued)      Address symbols

ID	Meaning
<i>od</i>	Outer displacement that is added after indirection occurs (MC68020 and MC68030 only). This is defined as an absolute expression. A word or long word is generated for the <i>od</i> as a function of its value, or for forward references as specified by the FORWARD directive. You may, however, explicitly control the generated size by using the syntax ( <i>od</i> ).W or ( <i>od</i> ).L.

Table 3-2 defines the syntax accepted for each addressing mode. Not all addressing modes are allowed for all machine instructions. The Motorola manuals cited at the beginning of this chapter tell you which addressing modes may be used with each instruction. Some of the modes have alternate syntactic forms, as shown. These alternate forms are discussed in "Ambiguities and Optimizations," given later in this chapter.

■ **Table 3-2**      Address syntax summary

Mode	Addressing mode	Effective address syntax
0	Data register direct	<i>Dn</i>
1	Address register direct	<i>An</i>
2	Address register indirect	( <i>An</i> )
3	Postincrement address register indirect	( <i>An</i> ) <sup>+</sup>
4	Predecrement address register indirect	-( <i>An</i> )
5	Address register indirect with 16-bit displacement	<i>d</i> ( <i>An</i> )
6	Indirect with indexing plus 8-bit displacement	<i>d</i> ( <i>An</i> , <i>Xn</i> )
6*	Indirect with indexing plus base displacement	( <i>bd</i> , <i>An</i> , <i>Xn</i> * <i>s</i> ) <i>bd</i> ( <i>An</i> , <i>Xn</i> * <i>s</i> )
6*	Indirect with preindexing	([ <i>bd</i> , <i>An</i> , <i>Xn</i> * <i>s</i> ], <i>od</i> )
6*	Indirect with postindexing	([ <i>bd</i> , <i>An</i> ], <i>Xn</i> * <i>s</i> , <i>od</i> )
70	Absolute word (16 bits)	<i>ae</i> ( <i>ae</i> ).W
71	Absolute long (32 bits)	<i>ae</i> ( <i>ae</i> ).L
72	PC-relative with displacement	<i>re</i> <i>d</i> (PC)

■ **Table 3-2** (continued) Address syntax summary

Mode	Addressing mode	Effective address syntax		
72	Literal (PC-relative with 16-bit displacement)**	<i>#ae</i>	<i>*(ae).W</i>	<i>*(ae).L</i>
73	PC-relative, indexing, 8-bit displacement	<i>d(Dn)</i>		<i>d(PC,Xn)</i>
73*	PC-relative, indexing, base displacement	<i>(bd,PC,Xn*s)</i> <i>bdXn*s)</i>		<i>bd(PC,Xn*s)</i>
73*	PC-relative with preindexing	<i>([bd,PC,Xn*s],od)</i>		
73*	PC-relative with postindexing	<i>([bd,PC],Xn*s,od)</i>		
74	Immediate	<i>#ae</i>		

\* Modes usable only with the MC68020 and MC68030

\*\* MPW Assembler only; not supported directly by the processor.

When writing the address forms shown in Table 3-2, you must follow these rules:

- You must write parameters in the order shown.
- The square brackets shown in Table 3-2 do not indicate optional parameters. You must write the brackets as shown.
- Expressions that specify immediate operands, literals, or absolute addresses may not contain any forward, undefined, or imported references.
- Expressions that specify immediate operands and literals, with the exception of absolute addresses, may contain SET variables and functions. You may follow the rules for absolute expressions in macro directives, explained in Chapter 5.
- Expressions that specify displacements may contain imported references but no SET variable or function references.
- Any expression involving an unpaired data reference or a forward reference to an undefined identifier is assumed to be relocatable.
- Wherever *Xn* is shown, it may be written as *Xn.W* or *Xn.L* to indicate either 16-bit or 32-bit indexing. If the size is omitted, 16-bit indexing (suffix *w*) is assumed.
- The scale factor *\*s* may be omitted. If omitted, a scale factor of 1 is assumed. If it is specified, *s* must be an absolute expression with the value 1, 2, 4, or 8. Values of 2, 4, and 8 are allowed only with the MC68020 and MC68030.

- The addressing form  $bd(A_n, X_n * s)$  with a scale factor  $s$  of 1, 2, 4, or 8 generates a brief MC68020 effective address format, while the form  $(bd, A_n, X_n * s)$  generates the full format. Similarly for the PC-relative forms:  $bd(PC, X_n * s)$  attempts to generate the brief format and  $(bd, PC, X_n * s)$  generates the full format. (Brief formats are possible only if  $bd$  is 8 bits.)
- The addressing form  $d(D_n)$  is equivalent to  $d(PC, D_n)$  and generates the effective address  $d-*(PC, D_n)$ . Similarly, the form  $bd(X_n * s)$  generates the effective address  $bd-*(PC, X_n * s)$ .
- When two registers appear in parentheses, if the leftmost could be either  $A_n$  or  $X_n$  (that is, if no explicit scaling or size is specified), then the base register  $A_n$  is assumed to be the leftmost and the second is assumed to be the index register  $X_n$ .
- Parameters may be omitted in the six additional MC68020/MC68030 modes. If a parameter is omitted, the comma preceding it, if any, must also be omitted. Omitted registers take on suppressed register values (0). Omitted displacements or displacements with the value 0 take on null values (also 0). Omitted parameters may result in ambiguous addressing modes. These are discussed next, in "Ambiguities and Optimizations." You can resolve these ambiguities by using zero-suppressed registers—registers whose values are treated as 0 during effective address calculations.
- Register mnemonics  $ZPC$ ,  $ZA0..ZA7$ , and  $ZD0..ZD7$  specify zero-suppressed registers. These symbols may be used to specify any allowable register in the six additional MC68020/MC68030 addressing modes. Using such mnemonics also explicitly forces one of the extended addressing modes, if omitting the specified registers would cause the Assembler to substitute a simpler mode. For the rules covering such substitutions, see the next section, "Ambiguities and Optimizations."
- Equates ( $EQU$  and  $SET$  directive statements) to absolute values (such as constants and registers) must be written before the equated symbols are used in the source text. When the Assembler encounters a symbol in an effective address, the Assembler first looks for the symbol in the code module's local symbol table. If it does not find the symbol there, it searches for it in the global symbol table.
- Displacements are always sign-extended to 32 bits by the processor. Because a number like  $\$FFF6$  could be incorrectly interpreted by a human (as 65526), it has been made illegal in the MPW Assembler. To specify an offset of -10, you must write either  $-10(A5)$  or  $-\$A(A5)$ .



- Addressing mode 71 (absolute long) may be optimized to addressing mode 70 (absolute word), depending on the value of your operand. Because \$FFFF in the upper word of a 32-bit operand is equivalent to sign-extension of a lower word that is between \$8000 and \$FFFF, and because the absolute word mode is more efficient than the absolute long mode, the Assembler automatically optimizes when it can, even if `OPT NONE` is in effect. Thus, if your absolute expression is 32 bits long and has \$FFFF (or \$0000) in its upper byte, matching the value of the high bit of the low word, the Assembler automatically generates an instruction that uses the absolute word addressing mode. If you want to force a specific size, use the alternate notation. For example, by writing `($FEDC).L` you can force the value of \$FEDC to 32 bits, padded with zeros, and by writing `($0FEDC).W` you can force the value to 16 bits (in this case, \$FEDC). Remember that this truncation proceeds without notice or warning, and produces \$FFFFFFEDC after the processor sign-extends it!
- With the Macintosh, all global data references are relative to the address in register A5. This means that mode 5 or 6 addresses referring to global data should specify A5 as the address register. With data record fields or imported templates, such a specification takes the form *record.field(A5)* or *record.field(A5, Xn)*, where *record* is a data module record identifier and *field* is a field identifier within the record. See the discussion of the Memory Manager in *Inside Macintosh*, Volume II, for a description of the use of register A5.
- ◆ *Note:* If you specify a data field reference without an explicit base register, the Assembler will assume register A5 and will change the addressing mode to mode 5, as appropriate.

---

### Ambiguities and optimizations

Under certain conditions, the Macintosh Assembler will transform the address syntax you write in your source text to a simpler form. It does this to remove ambiguities, reduce object code, and improve execution speed. These automatic transformations are summarized in Table 3-3.

■ Table 3-3 Effective address transformations

Original form	Condition for optimization	Optimized form
$(bd, An, Xn*s)$	Size of $bd \leq 8$ bits	$bd(An, Xn*s)$
$(An, Xn*s)$	Omitted $bd$ ( $bd = 0$ )	$0(An, Xn*s)$
$(bd, PC, Xn*s)$	Size of $bd \leq 8$ bits	$bd(PC, Xn*s)$
$(bd, An)$	Size of $bd \leq 16$ bits	$bd(An)$
$(bd, PC)$	Size of $bd \leq 16$ bits	$bd(PC)$
$d(An)$	$d = 0$	$(An)$

Here are the rules by which the Assembler performs automatic address transformation during assembly:

- The syntax for the mode 6 MC68020 extended addressing form,  $(bd, An, Xn*s)$ , is identical to the mode 2 addressing mode,  $(An)$ , if the displacement and index are omitted. If just the index is omitted, and the displacement is 16 bits or less, then the mode 6 form,  $(bd, An)$ , is identical to mode 5,  $d(An)$ . Even when the index is specified, if the displacement is eight bits or less then the mode 6 form,  $(bd, An, Xn)$ , is identical to the form  $d(An, Xn)$ . The Assembler resolves these ambiguities by selecting the more efficient forms.
- A similar situation exists for the mode 73 form,  $(bd, PC, Xn*s)$ . It is identical to mode 72,  $d(PC)$ , when the index is omitted and the displacement is 16 bits or less, and to mode 73,  $d(PC, Xn)$ , when the displacement is eight bits or less. As in the mode 6 cases, the Assembler chooses the more efficient mode 72 form.
- The Assembler optimizes each address before it checks to see if it is an MC68020 address. If an MC68020 address is optimized to an MC68000 form, the Assembler will accept it even when the target microprocessor is not the MC68020. Hence you can use MC68020 forms to write MC68000 addresses if you really want to, provided they meet the criteria for optimization.
- Some optimizations are made by detecting when it is possible to use the brief format extension word instead of the full format extension word in the extended addressing modes of the MC68020 and MC68030. The full format extension word is always used when OPT NONE is selected.

Normally, the Assembler tries to use the shorter and more efficient form when interpreting the foregoing addressing modes. If you want to preserve the extended address form, use the `OPT` directive described under "Assembly Options" in Chapter 4 to suppress transformation. In this case, remember that preindexing is more efficient than postindexing, and using an index register is more efficient than using a base register with displacement for indirect modes.

If you want to override automatic mode transformations with an individual instruction and explicitly force a specific mode, use zero-suppressed registers (`ZPC`, `ZA0..ZA7`, `ZD0..ZD7`).

- ◆ *Note:* Remember that when the program counter is zero-suppressed (`ZPC`), its displacement is assumed to be absolute and hence is not offset from the current location-counter (`PC`) value.

---

### Forward-reference addressing

The size of the displacement values in the various modes of effective addresses can be 8, 16, or 32 bits depending on the value, the mode, and the processor. When you use imported or forward-referenced identifiers in addresses, and there is no other way to determine their size, the Assembler assumes default sizes for the various displacements. All such default actions may be overridden with the `BRANCH` and `FORWARD` assembly-control directives.

The `BCC`, `BSR`, `BRA`, `FBCC`, and `PBCC` instructions contain 8-bit, 16-bit, and 32-bit PC-relative displacements without any explicit mode indication in their syntax. The 32-bit displacement is available only in the MC68020 and MC68030, and the `FBCC` and `PBCC` instructions are limited to 16-bit and 32-bit forms. The Assembler assumes a 16-bit forward-referenced offset, unless a period and suffix `s` or `L` is written after the mnemonic. The `BRANCH` directive allows you to change the default size.

The `FORWARD` directive controls the default size for all other forward-referenced displacement encodings—offsets, base displacements, and outer displacements. The default size is 16 bits; you can change this to 32 bits with the MC68020 and MC68030 only.

---

## Registers

The addressing modes defined in Table 3-2 use the standard MC68000 address registers (A0 through A7), data registers (D0 through D7), and program counter (PC). Besides these, the other processors and coprocessors supported by the MPW Assembler contain additional registers, which may be named in some instructions and not in others. Table 3-4 lists all the registers recognized by the Assembler, including those already discussed. Refer to the appropriate Motorola manuals, listed in the preface of this manual, for the exact formats and uses of these registers.

■ **Table 3-4** Registers

Designation	Usage
<b>MC68000, MC68010, MC68020 and MC68030</b>	
D0..D7	Data registers
A0..A7	Address registers
A7, SP	The current stack pointer
SR	Status register
USP	User stack pointer
MSP, SSP	Master stack pointer
PC	Program counter
<b>MC68010, MC68020, and MC68030 only</b>	
SFC	Source function code register
DFC	Destination function code register
VBR	Vector base register
<b>MC68020 and MC68030 only</b>	
ISP	Interrupt stack pointer
CACR	Cache control Condition code register
CAAR	Cache address register
ZPC	Zero-suppressed program counter
ZA0..ZA7	Zero-suppressed address registers
ZD0..ZD7	Zero-suppressed data registers

■ **Table 3-4** (continued)      Registers

<b>Designation</b>	<b>Usage</b>
<b>MC68030 only</b>	
TT0..TT1	Transparent translation control registers
MMUSR	Memory Management Unit Status Register
CRP	CPU root pointer register
SRP	Supervisor root pointer register
TC	Translation control register
<b>MC68851 only</b>	
CRP	CPU root pointer register
SRP	Supervisor root pointer register
DRP	DMA root pointer register
PCSR	PMMU cache status register
TC	Translation control register
AC	Access control register
CAL	Current access level register
VAL	Validate access level register
SCC	Stack change control register
PSR	PMMU status register
BAC0..BAC7	Breakpoint acknowledge control registers
BAD0..BAD7	Breakpoint acknowledge data registers
<b>MC68881 and MC68882 only</b>	
FP0..FP7	Floating-point data registers
FPCR	Floating-point control register
FPSR	Floating-point status register
FPIAR	Floating-point instruction address register

Any of the register names listed in Table 3-4 may be equated to other identifiers by using the `EQU` and `SET` directives. If you do this, make sure that you write the equates before you use the new symbols. When the Assembler encounters a symbol in an effective address position that may be a register, it first looks for the symbol in the code module's local symbol table. If it doesn't find it there, it searches the global symbol table.

---

## Special address formats

Most MC68xxx instructions contain two effective addresses separated by a comma. The first address is called the **source** and the second the **destination**. The instructions generally cause an operation to be performed on the source, possibly in combination with the destination, and place a result in the destination. However, there are some exceptions to this format. This section gives the syntax rules for such exceptions.

In the following sections, *ea* represents any effective address format that may legally be used with the instruction being discussed.

---

### MC68xxx instructions

#### MOVEM: Multiple moves

```
MOVEM . size    rlist, ea
MOVEM . size    ea, rlist
size ::= W | L
```

The MOVEM instruction takes a **register list**, *rlist*, as either a source or destination. The register list syntax is as follows:

- *Rm-Rn* designates registers *Rm* through *Rn* (where  $m \leq n$ , and *Rm* and *Rn* are both A registers or both D registers).
- *Ri/Rj/Rk...* designates registers *Ri*, *Rj*, *Rk...* where each term is an A register, a D register, or a range *Rm..Rn*.

Here are two examples:

Example	Meaning
D0-D1/A3	D0, D1, and A3
D2-D4/A1-A2/D7	D2, D3, D4, A1, A2, and D7

---

## MC68020 instructions

For details of the syntax of these instructions, see the Motorola *MC68020 32-Bit Microprocessor User's Manual*.

### MULS and MULU: Signed and unsigned multiplication

MULS.L	<i>ea, Dl</i>	32 x 32 → 32
MULS.L	<i>ea, Dh:Dl</i>	32 x 32 → 64
MULU.L	<i>ea, Dl</i>	32 x 32 → 32
MULU.L	<i>ea, Dh:Dl</i>	32 x 32 → 64

In these syntax diagrams, *Dl* designates the low-order register (*l* = 0..7) and *Dh* the high-order register (*h* = 0..7). These instructions support 32-bit multipliers and a 32-bit or 64-bit product, as shown by the comments in the far right column.

### DIVS and DIVU: Signed and unsigned division

DIVS.L	<i>ea, Dq</i>	32/32 → 32q
DIVS.L	<i>ea, Dr:Dq</i>	64/32 → 32r:32q
DIVU.L	<i>ea, Dq</i>	32/32 → 32q
DIVU.L	<i>ea, Dr:Dq</i>	64/32 → 32r:32q

In these syntax diagrams, *Dq* designates the quotient register (*q* = 0..7) and *Dr* the remainder register (*r* = 0..7). These instructions support a 64-bit dividend, a 32-bit quotient, and a 32-bit remainder, as shown by the comments in the far right column.

### TDIVS and TDIVU: Truncated signed and unsigned division

TDIVS.L	<i>ea, Dq</i>	32/32 → 32q
TDIVS.L	<i>ea, Dr:Dq</i>	32/32 → 32r:32q
TDIVU.L	<i>ea, Dq</i>	32/32 → 32q
TDIVU.L	<i>ea, Dr:Dq</i>	32/32 → 32r:32q

In these syntax diagrams, *Dq* designates the quotient register (*q* = 0..7) and *Dr* the remainder register (*r* = 0..7). These instructions divide two 32-bit values and return either a quotient and a remainder or just a quotient, as shown by the comments in the far right column.

- ◆ *Note:* The current edition of the Motorola MC68020 user's manual uses the mnemonic `DIVSL.L` to refer to these instructions.

## PACK and UNPK: Packing and unpacking

```
PACK      -(Ax) , -(Ay) , #adjustment
PACK      Dx, Dy, #adjustment
UNPK      -(Ax) , -(Ay) , #adjustment
UNPK      Dx, Dy, #adjustment
```

These instructions pack and unpack BCD digit formats between the source (*x*) and destination (*y*) registers. The *adjustment* is a 16-bit absolute expression added to the source value to allow character translation. This expression follows the same rules as those for immediate operands.

## CAS and CAS2: Comparing and swapping

```
CAS .size      Dc, Du, ea
CAS2 .size     Dc1:Dc2, Du1:Du2, (Rn1) : (Rn2)
size ::= B | W | L
```

These instructions are most easily explained as if they were a sequence of pseudo-Pascal statements:

```
CAS:  IF Dc=ea^
      THEN                               {We have a match}
        ea^ := Du                        {Copy Du to ea^}
      ELSE                               {No match}
        Dc := ea^;                       {Copy ea^ to Dc}

CAS2: IF (Dc1 = Rn1^) AND (Dc2 = Rn2^)
      THEN                               {We have a match}
        BEGIN                             {Set destination}
          Rn1^ := Du1;                     {Copy Du1 to Rn1^}
          Rn2^ := Du2;                     {Copy Du2 to Rn2^}
        END
      ELSE                               {No match}
        BEGIN
          Dc1 := Rn1^;                     {Copy Rn1^ to Dc1}
          Dc2 := Rn2^;                     {Copy Rn2^ to Dc2}
        END;
```

Both instructions operate the same way, except that CAS2 operates on two sets of registers simultaneously while CAS operates on only one set of registers.



## Bit field instructions

<b>BFCHG</b>	<i>ea</i> { ' <i>offset: width</i> ' }
<b>BFCLR</b>	<i>ea</i> { ' <i>offset: width</i> ' }
<b>BFEXTS</b>	<i>ea</i> { ' <i>offset: width</i> ' }, <i>Dn</i>
<b>BFEXTU</b>	<i>ea</i> { ' <i>offset: width</i> ' }, <i>Dn</i>
<b>BFFFO</b>	<i>ea</i> { ' <i>offset: width</i> ' }, <i>Dn</i>
<b>BFINS</b>	<i>Dn</i> , <i>ea</i> { ' <i>offset: width</i> ' }
<b>BFSET</b>	<i>ea</i> { ' <i>offset: width</i> ' }
<b>BFTST</b>	<i>ea</i> { ' <i>offset: width</i> ' }

These instructions operate on a string of consecutive bits in a bit array. In the syntax given, the braces and colon must be included as shown. A comma may be used in place of the colon. The *offset* and *width* parameters must be either data registers or absolute expressions. If they are expressions, they must follow the same rules as those for immediate operands.

## Tcc and TPcc: Trap on condition

```
TCC
TPcc.size      #ae
size ::= W | L
```

These instructions are the same as described in the MC68020 user's manual except that the single mnemonic, `TRAPcc`, has been changed to two mnemonics, `TCC` and `TPcc`. `TCC` is used for the parameterless form, while `TPcc` is used when an immediate data operand is specified.

## Assembler control

Source text written for the MC68020 processor must contain the following directive before any MC68020 operations:

```
[macro-label]          MACHINE          MC68020
```

This directive tells the Assembler to process all subsequent code to run on the MC68020.

---

## The MC68030 processor

The MC68030 processor is a single chip that combines most of the capabilities of the MC68020 processor with some, but not all, of the capabilities of the MC68851 Paged Memory Management Unit coprocessor. It is discussed in full detail in the Motorola *MC68030 Enhanced 32-Bit Microprocessor User's Manual*.

## Assembler control

Source text written for the MC68030 processor must contain the following directive before any MC68030 operations:

```
[macro-label]          MACHINE          MC68030
```

This directive tells the Assembler to process all subsequent code to run on the MC68030.

- ▲ **Warning**      The source text must not contain an MC68851 directive as well; if it does, the Assembler reports an error. However, the source text may include an MC68881 directive. ▲

After a MACHINE MC68030 directive, the &SETTING function will return a value of MC68030 when its operand is MACHINE. The &SETTING function is described in Chapter 5.

## MC68020 statements you can use

In source text intended for the MC68030 processor, you may use any of the instructions and directives valid for the MC68020 except the CALLM and RTM instructions. These two instructions may be used only with the MC68020 processor.

## MC68851 instructions you can use

The MC68030 processor contains only six of the MC68851 registers (in addition to the MC68020 registers), and can execute only some of the MC68851 coprocessor instructions. The registers are listed in Table 3-5; the valid coprocessor instructions are listed in Table 3-6.

■ **Table 3-5**      MC68851 registers in the MC68030

Designation	Usage
CRP	CPU root pointer register
SRP	Supervisor root pointer register
MMUSR.	PMMU status register (PSR in the MC68851)
TC	Translation control register
TT0..TT1	Transparent translation control registers

■ Table 3-6 MC68851 instructions valid for the MC68030

Opcode	Operand format	Sizes
PFLUSH	<i>fc, #ae, ea</i>	
PFLUSHA		
PLOADR	<i>fc, ea</i>	
PLOADW	<i>fc, ea</i>	
PMOVE	<i>PMMU-reg, ea</i>	depends on PMMU-reg
PMOVE	<i>ea, PMMU-reg</i>	depends on PMMU-reg
PTESTR	<i>fc, ea, #ae, An</i>	
PTESTW	<i>fc, ea, #ae, An</i>	
The MC68030 also allows one instruction that is not valid for the MC68851:		
PMOVEFD	<i>ea, PMMU-reg</i>	depends on PMMU-reg

The mask, *#ae* in several of the instructions in Table 3-6, is a three-bit absolute expression and is stored in the instruction. The function code, *fc* in several of the instructions, may be specified as follows:

*fc* ::= *#ae* (specified as three bits in the command word, absolute expression)  
*Dn* (contained in the lower three bits of *Dn*)  
*SFC* (contained in the processor's source function register)  
*DFC* (contained in the processor's destination function code register)

The PMMU registers, mentioned in the PMOVE instruction, are listed in Table 3-5.

The root pointer registers in the MC68030 contain double long words, 64 bits long. The PMOVE instruction, which references these registers, accepts immediate effective addresses; hence for *ea* you can use *#ae* (mode 74, Table 3-2). If you do this, however, the Assembler will convert the 32-bit effective address to a 64-bit value by filling it on the left with 32 zero bits. It will also issue a warning, because it does not support 64-bit values. You can avoid this limitation by defining a constant with two DC.L directives, then referencing them in your PMOVE instruction.

---

## MC68881 and MC68882 instructions

For details of the syntax of these instructions, see the Motorola *MC68881/MC68882 Floating-Point Coprocessor User's Manual*.

### FMOVEM with explicit register lists

```
FMOVEM .size    fp-rlist, ea
FMOVEM .size    ea, fp-rlist
size ::= L | X
```

The FMOVEM instruction takes a floating-point register list, *fp-rlist*, as either a source or a destination. Here are the rules of register list syntax:

- $FPm-FPn$  designates floating-point registers  $FPm$  through  $FPn$  where  $m \leq n$ .
- $FPi/FPj/FPk...$  designates registers  $FPi, FPj, FPk...$  Each term is either a single register  $FPn$  or a range  $FPm..FPn$ .
- $FPCR/FPSR/FPIAR$  designates the floating-point control registers FPCR, FPSR, and FPIAR, in any order. You cannot combine these registers with other registers in a register list.

Here are two examples:

Example	Meaning
$FP0-FP3/FP7$	FP0, FP1, FP2, FP3, and FP7
FPCR/FPSR	Control registers FPCR and FPSR

### FMOVE with packed BCD data

```
FMOVE .P        FPn, ea
FMOVE .P        FPn, ea {'#k'}
FMOVE .P        FPn, ea {'Dn'}
```

When writing this instruction you must include the braces as shown. The numerical expression inside the braces is the k-factor, which tells the MC68881 coprocessor in what format to construct the resulting decimal string. It may be expressed either dynamically (as the value in register  $Dn$ ), as an absolute expression preceded by the pound sign (#), or by default. For an explanation of k-factors, see the discussion of FMOVE in the Motorola *MC68881/MC68882 Floating-Point Coprocessor User's Manual*.

### FSINCOS: Simultaneous sine and cosine

```
FSINCOS .size    ea, FPC:FPS
FSINCOS .X      FPM, FPC:FPS
size ::= B | W | L | S | D | X | P
```

*FPC* is the floating-point register holding the cosine result. *FPS* is the floating-point register holding the sine result. *FPM* is the floating-point register holding the source value.

### FTCC and FTPCC: Floating-point trap on condition

```
FTCC
FTPCC.size      #ae
size ::= W | L
```

These instructions are the same as *FTRAPCC*, described in the Motorola MC68881/MC68882 user's manual, except that the two mnemonics *FTCC* and *FTPCC* have been substituted for *FTRAPCC*. You write *FTCC* for the form without parameters and *FTPCC* for the form with an operand.

### FTEST: Test operand and set floating-point condition codes

```
FTEST .size     ea
FTEST .X       FPn
size ::= B | W | L | S | D | X | P
```

The *FTEST* instruction is the same as *FTST*, described in the Motorola MC68881/MC68882 *Floating-Point Coprocessor User's Manual*. The mnemonic was changed to *FTEST* to avoid ambiguity with the *FTCC* instruction using a signaling true (*ST*) conditional predicate.

---

## MC68851 instructions

If your source text contains code for the MC68851 PMMU coprocessor, you may use special operand formats with the instructions listed in Table 3-7. For details of the syntax of these instructions, see the Motorola *MC68851 Paged Memory Management Unit User's Manual*.

■ Table 3-7 Special MC68851 operand formats

Opcode	Operand format	Sizes	Notes
PBcc.size	label	W   L	
PDBcc.size	Dn, label	W	
PFLUSH	fc, #ae[, ea]		1
PFLUSHA			
PFLUSHS	fc, #ae[, ea]		1
PFLUSHR	ea	D	4
PLOADR	fc, ea		1
PLOADW	fc, ea		1
PMOVE	PMMU-reg, ea	B   W   L   D	2
PMOVE	ea, PMMU-reg	B   W   L   D	2,4
PRESTORE	ea		
PSAVE	ea		
PSCC	ea	B	
PTESTR	fc, ea, #ae[, An]		1
PTESTW	fc, ea, #ae[, An]		1
PTCC			3
PTPCC	#ae	W   L	3
PVALID	VAL, ea	L	2
PVALID	An, ea	L	

1. The function code is defined as follows:

```

fc ::= #ae (specified as three bits in the command word)
      Dn (contained in the lower three bits of Dn)
      SFC (contained in the processor's source function register)
      DFC (contained in the processor's destination function code register)

```

2. The MC68851 registers are listed in Table 3-4.

3. The Assembler recognizes the instruction mnemonics PTCC and PTPCC in place of the Motorola mnemonic PTRAPCC. Use PTCC for the form without parameters and PTPCC for the form with an immediate data operand.

4. The root pointer registers in the MC68851 contain double long words, 64 bits long. The FLUSHR and PMOVE instructions, which reference these registers, accept immediate effective addresses; hence for ea you can use #ae (mode 74, Table 3-2). If you do this, however, the Assembler converts the 32-bit effective address to a 64-bit value by filling it on the left with 32 zero bits. It also issues a warning, because it does not support 64-bit values. You can avoid this limitation by defining a constant with two DC.L directives, then referencing them in your FLUSHR or PMOVE instruction.

5. The label in the PBcc.size and PDBcc.size instructions must obey the rules for relocatable expressions.

---

## Literals

Frequently, it is necessary to push the address of a constant value onto the stack. Unfortunately, in the MC68xxx instruction set, the effective addressing modes for the PEA and LEA instructions do not permit immediate data. Nonetheless, these instructions are used quite often for passing parameters to subroutines, particularly for passing string addresses to Macintosh ROM routines. Hence the MPW Assembler allows you to specify immediate data to PEA and LEA instructions. As used here, an absolute expression (with no forward, imported, or undefined references) or a string is called a **literal**.

The syntax for writing PEA and LEA instructions with literals is as follows:

PEA	# <i>data</i>	Pushes address of <i>data</i>
LEA	# <i>data</i> , A <i>n</i>	Loads address of <i>data</i>
PEA	#'MyConstant'	Pushes address of 'MyConstant'

This is functionally equivalent to the following:

```
PEA    L1
LEA    L1, An
PEA    ConstAddr
---
ALIGN  2      ; Must be on word boundary
L1     DC .size data
ConstAddr DC .size 'MyConstant'
```

The *size* qualifier is B or W or L, corresponding to the *data* size or to the explicit specification of the literal.

The Assembler creates a PC-relative mode-72 address when processing PEA and LEA instructions.

All literals encountered during the assembling of a code module are accumulated in an area called the **literal pool**. Multiple references to the same literal address only generate one instance of the literal in the literal pool; duplicate literals are not generated. The literal pool is attached to the end of the code module as part of the code.

When a string literal is generated, it may be any one of the three formats for character strings—an as-is string, a C string, or a Pascal string. The Assembler determines which format to use by the current setting of the STRING directive at the time the literal is placed in the literal pool by the PEA or LEA instruction. A STRING directive setting at the end of the module has no effect on the format of strings in the literal pool.

If an absolute expression is used to generate a literal, the size of the literal depends on its value, as follows:

- Values between -32767 and (unsigned) 65535 are created as word-size literals.
- All other integer values are created as long-word literals.
- Floating-point values are created as extended (12-byte) literals. You may use such values only if the `MC68881` directive is in force.

Because both strings and absolute expressions may be used as literals, the Assembler may interpret an absolute literal incorrectly if its first symbol is a string constant. To force the Assembler to treat a literal as absolute, enclose it in parentheses.

The Assembler lets you override the implicit sizing of numeric literals by explicitly specifying their size. This is done by extending the literal syntax, as follows:

PEA	<code> #(ae) .W</code>	Immediate word data for PEA
PEA	<code> #(ae) .L</code>	Immediate long-word data for PEA
PEA	<code> #(ae) .S</code>	Immediate single-precision data for PEA
PEA	<code> #(ae) .D</code>	Immediate double-precision data for PEA
PEA	<code> #(ae) .X</code>	Immediate extended data for PEA
PEA	<code> #(ae) .P</code>	Immediate packed BCD data for PEA
LEA	<code> #(ae) .W, An</code>	Immediate word data for LEA
LEA	<code> #(ae) .L, An</code>	Immediate long-word data for LEA
LEA	<code> #(ae) .S, An</code>	Immediate single-precision data for LEA
LEA	<code> #(ae) .D, An</code>	Immediate double-precision data for LEA
LEA	<code> #(ae) .X, An</code>	Immediate extended data for LEA
LEA	<code> #(ae) .P, An</code>	Immediate packed BCD data for LEA

- ◆ *Note:* Single, double, extended, and packed BCD data can be used only if the `MC68881` directive is in force.

Enclosing the literal in parentheses and following it with a size qualifier (such as `w`) establishes its size. The value of the literal must always lie within the range specified. Size qualifiers are described in Table 2-1.



## Chapter 4 Assembler Directives

DIRECTIVES ARE INSTRUCTIONS TO THE MPW ASSEMBLER to perform specific operations during assembly. ■

### *Contents*

Assembler directives	59
Code and data module definitions	59
Symbol definitions	59
Data definitions	59
Template definitions	59
Linker and scope controls	60
Assembly options	60
Location-counter controls	60
File controls	60
Listing controls	60
Directive formats	61
Code and data module definitions	62
PROC and ENDPROC: Define procedure code module	62
FUNC and ENDFUNC: Define function code module	63
MAIN and ENDMAIN: Define main program code module	63
RECORD and ENDR: Define a data module	64
INCREMENT and DECREMENT	65
MAIN	66
CODE and DATA: Switch between code and data	67
END: End the assembly	67
Symbol definitions	68
EQU and SET: Name constants and registers	68
REG and FREG: Name register list	70
OPWORD: Name machine instruction	71
Data definitions	72
DC and DCB: Place constants in code or data	73
DS: Define storage area	75

- Template definitions 76
  - RECORD and ENDR: Define a template 76
    - Using templates as data types 81
  - WITH and ENDWITH: Supply RECORD name qualification 82
- Linker and scope controls 84
  - EXPORT and ENTRY: Expand scope of entry points 85
  - IMPORT: Identify external entry points 87
  - CODEREFS and DATAREFS: Control name linking 88
    - Code-to-code references 89
    - Code-to-data references 90
    - Data-to-code references 90
    - Data-to-data references 91
  - SEG: Specify current code segment 92
  - COMMENT: Place a comment in object file 93
- Assembly options 93
  - MACHINE: Specify target machine 93
  - MC68881: Assemble MC68881/MC68882 coprocessor instructions 94
  - MC68851: Assemble MC68851 coprocessor instructions 95
  - STRING: Specify string format 95
  - BRANCH and FORWARD: Resolve forward branches 96
  - OPT: Specify level of code optimization 97
  - CASE: Specify treatment of lowercase letters 98
    - Writing register names 99
  - BLANKS: Control acceptance of blanks in operand field 99
- Location-counter controls 100
  - ALIGN: Align location counter 100
    - Special cases 101
  - ORG: Set location counter 102
- File controls 103
  - File search rules 104
  - INCLUDE: Take source text from another file 104
  - DUMP and LOAD: Write and read symbol table files 105
  - ERRLOG: Specify error log file 106
- Listing controls 107
  - PAGESIZE: Specify listing page size 107
  - TITLE: Specify title line for listing 108
  - PRINT: Control listing information 108
  - EJECT: Start new listing page 111
  - SPACE: Insert blank line in listing 111

---

## Assembler directives

A number of MPW Assembler directives (RECORD, PROC, EXPORT, SEG, and so on) were mentioned in Chapter 2. This chapter covers them and others in detail. The discussion is organized into these groups:

### Code and data module definitions

PROC	Begin a procedure code module
ENDPROC	End a procedure code module
FUNC	Begin a function code module
ENDFUNC	End a function code module
MAIN	Begin a main program code module
ENDMAIN	End a main program code module
RECORD	Begin a data module
ENDR	End a data module
CODE	Switch assembly from data to code
DATA	Switch assembly from code to data
END	End the whole assembly

### Symbol definitions

EQU	Assign a permanent value to a symbol
SET	Assign a temporary value to a symbol
REG	Assign an identifier to a processor register list
FREG	Assign an identifier to an MC68881 register list
OPWORD	Assign an identifier to an opcode

### Data definitions

DC	Place constants in a code or data module
DCB	Place a block of constants in a code or data module
DS	Define a storage area

### Template definitions

RECORD	Begin a record template definition
ENDR	End a record template definition
WITH	Begin default record identifier qualification
ENDWITH	End default record identifier qualification

## Linker and scope controls

EXPORT	Make entry points accessible in other assemblies
ENTRY	Make local entry-points global
IMPORT	Identify entry points declared externally
CODEREFS	Control the linking of code-to-code references
DATAREFS	Control the linking of data-to-code and data-to-data references
SEG	Specify the current code segment
COMMENT	Place a comment in the object file

## Assembly options

MACHINE	Identify the target microprocessor model
MC68881	Control the assembly of floating-point coprocessor instructions
MC68851	Control the assembly of PMMU coprocessor instructions
STRING	Control the encoding of string constants
BRANCH	Control the encoding of branch instructions
FORWARD	Control the encoding of forward references
OPT	Control the level of code optimization
CASE	Control the treatment of lowercase letters in identifiers
BLANKS	Control the treatment of spaces and tabs in the operand field

## Location-counter controls

ALIGN	Advance the location counter to the next multiple of a value
ORG	Set the value of the location counter

## File controls

INCLUDE	Insert source text from another file
DUMP	Write the current global symbol table to a file
LOAD	Read a file into the current global symbol table
ERRLOG	Create an error-listing file

## Listing controls

PAGESIZE	Specify the listing page size
TITLE	Define a title for the listing header
PRINT	Control miscellaneous listing options
EJECT	Start a new page in the listing
SPACE	Insert blank lines in the listing

In addition to the directives listed above, the MPW Assembler supports directives for macro definition and expansion, macro variables, and conditional assembly. These are described in Chapters 5, 6, and 7.

---

## Directive formats

Macintosh directives follow the general format for Assembler statements. You write them in four fields, separated by spaces or tabs, as described in Chapter 2 under "Machine Instruction Syntax":

Label field	Operation field	Operand field	Comment field
[ <i>identifier</i> ]	<i>directive name</i>	[ <i>directive parameters</i> ]	[ <i>comments</i> ]

The identifier in the label field may be required by the directive or may be an optional macro label. If you include a macro label, you can reference the directive from macro statements as described under "GOTO, IF...GOTO, and Macro Labels: Branching" in Chapter 7.

If the directive does not require a label or allow an optional macro label, you cannot include a label with it.

The directive name specifies which directive the Assembler executes. It is always required. The Assembler makes no distinction between uppercase and lowercase letters in directive names.

The operand field contains the directive's parameters, if any. Such parameters may be either required or optional, depending on the directive.

You can include a comment with a directive, writing it after all the directive's required parameters (if any). With directives that have no parameters or have only optional parameters, you can still include comments even if you don't specify any parameters. Use the standard convention of placing a semicolon in the operand field, following it with the comment field. Directive comments are ignored by the Assembler.

- ◆ *Note:* In the remainder of this chapter and in Chapters 5, 6, and 7, directive syntax is usually defined by means of syntax diagrams. You can find the rules for interpreting these diagrams under "Notation Conventions" in the Preface.

---

## Code and data module definitions

"Source Text Structure" in Chapter 2 describes how Macintosh object files are built from code and data modules. The directives described in this section delimit the code and data parts of your source text and tell the Assembler how to apply the statements in each part to specific modules. They are the following:

PROC	Begin a procedure code module
ENDPROC	End a procedure code module
FUNC	Begin a function code module
ENDFUNC	End a function code module
MAIN	Begin a main program code module
ENDMAIN	End a main program code module
RECORD	Begin a data module
ENDR	End a data module
CODE	Switch assembly from data to code
DATA	Switch assembly from code to data
END	End the whole assembly

---

### PROC and ENDPROC: Define procedure code module

```
[ name]          PROC          [ { ENTRY } ]
                               [ { EXPORT } ]
                               statements
[macro-label]    ENDP[ROC]
```

A PROC directive in your source text marks the beginning of a code module. The code module extends from the PROC directive until the next ENDPROC, or until the start of the next code module (PROC, FUNC, or MAIN), the next data module (RECORD), or the end of the assembly (END). You can write ENDPROC as ENDP. Code modules are the only places where you can write machine instruction statements.

If you write a name in the label field of a PROC directive, it becomes the identifier of the code module that begins there. The identifier is global to the assembly file. If you do not provide an identifier, you must define entry points inside the module by using ENTRY or EXPORT directives. ENTRY and EXPORT are described later in this chapter.

You can declare the code module itself as EXPORT in two ways: by specifying the module identifier in an EXPORT directive before you define the module or by writing EXPORT as an operand in the PROC directive itself. In either case the PROC directive that begins an EXPORT code module must include an identifier in its label field.

If you do not declare a code module as `EXPORT`, the Assembler declares it as `ENTRY` by default. For clarity of documentation, you may explicitly declare it as `ENTRY` in the `PROC` directive or specify its identifier in an `ENTRY` directive before defining the module. The latter technique is useful if you need to make a forward reference to the module.

You can declare the identifier of a procedure code module as `MAIN` by using a previous `ENTRY` or `EXPORT` directive. This has the same effect as declaring it with the `MAIN` directive described later in this chapter.

Labels defined inside a code module are local to that module. The only way to make these labels accessible to other modules is to declare them as `EXPORT` or `ENTRY` inside the module.

---

### **FUNC and ENDFUNC: Define function code module**

```
[name]          FUNC          [ { ENTRY } ]
                               [ { EXPORT } ]
                               statements
[macro-label]   ENDF[UNC]
```

`FUNC` and `ENDFUNC` act exactly the same as `PROC` and `ENDPROC`. They are included for documentation purposes only, so that you can indicate that the code module is a function rather than a procedure. You can write `ENDFUNC` as `ENDF`.

---

### **MAIN and ENDMAIN: Define main program code module**

```
[name]          MAIN          [ { ENTRY } ]
                               [ { EXPORT } ]
                               statements
[macro-label]   ENDMAIN
```

`MAIN` and `ENDMAIN` act exactly like `PROC` and `ENDPROC`, except that they declare the code module that they define as the main program. The first executable statement of that code module becomes the execution entry point for the whole program.

To declare a code module statement other than the first executable statement as a main entry point, use a previous `ENTRY` or `EXPORT` directive. `ENTRY` and `EXPORT` are described later in this chapter.

An assembly, including all its linked parts, may have only one main program module or main entry point.

- ◆ *Note:* If your program contains one or more data modules containing DC or DCB directives, you must link it with the library file Runtime.o, which contains the data initialization routine `_DataInit`. If your main code module is written in assembly language, its first executable statement must be a call (JSR) to the entry point `_DataInit`. This entry point must also be declared as IMPORT. After returning from `_DataInit`, your program may unload the segment `%A5Init` that contains it, by calling the Macintosh routine `UnloadSeg`. If your main program is written in C or Pascal, no explicit call to `_DataInit` is required, because the run-time libraries for C and Pascal automatically take care of data initialization.

---

## RECORD and ENDR: Define a data module

```
[name]          RECORD      [ { ENTRY } ] [ { INCR[EMENT] } ]
                               [ { EXPORT } ] [ { DECR[EMENT] } ]
                               directives
[macro-label]   ENDR
```

RECORD and ENDR let you delimit and name a data module. The data module extends from the RECORD directive until the next ENDR or until the start of the next code module (PROC, FUNC, MAIN), the next data module or template (RECORD), or the end of the assembly (END). RECORD and ENDR act like PROC and ENDPROC, but define a data module instead of a code module.

- ◆ *Note:* RECORD and ENDR are also used to define templates. This usage is described below under "Template Definitions."

Data modules may contain only directives. Some of these directives—ORG, ALIGN, DC, DCB, and DS—define data fields. Others, such as symbol definitions, define data within the fields.

Every data module must have an identifier. This is because the identifier is used to qualify the module's field identifiers when they are accessed from code modules. Unlike labels in code modules, the field labels in data modules may be accessed by all code modules that follow them in the source text file. You can also make them accessible to other files by including EXPORT directives inside the module. Conversely, you must define a data module before accessing any of its fields in the same file.

You can access a field in a data module by an identifier of the form *mod.field*, where *mod* is the identifier of the data module and *field* is a label inside the module.



Data modules may be declared as `EXPORT` or `ENTRY` just like code modules. You can either specify the module identifier in an `EXPORT` or `ENTRY` directive before defining the module, or include `EXPORT` or `ENTRY` as an operand in the `RECORD` directive itself. If you do not specify one or the other, the Assembler declares the data module as `ENTRY`.

The MPW Linker collects all global data modules so that they may be loaded as a group by the Segment Loader. It loads them just below the application parameters, pointed to by `A5`. Thus all global data modules are accessed relative to `A5`, with negative offsets determined by the Linker. The implied base register for qualified field references is `A5`; it need not be specified in machine instructions unless indexing is used. See the discussion of the Memory Manager in *Inside Macintosh*, Volume II, for further details about `A5`.

As with code modules, data modules have their own location counter; it points to the next available data location in the data module. In any data module, the value of this counter may range from `-32768` to `+32767`.

### **INCREMENT and DECREMENT**

In code modules, each machine instruction is executed immediately after the one preceding it. Thus the location counter for a code module is incremented for each instruction. Data module location counters act similarly, except that you can choose whether they increment or decrement.

`INCREMENT` is the default action for any data module location counter. If you specify it in a `RECORD` directive or omit the parameter altogether, the Assembler increments the resulting data module's location counter by the size of each piece of data after that data is allocated. The location counter therefore always points to the lowest address of the next piece of data.

If you specify `DECREMENT` in a `RECORD` directive, the Assembler allocates data in the data module in the reverse direction. This corresponds to the allocation algorithm of Pascal. The location counter is first decremented by the size of each piece of data, before it is allocated; hence, each piece of data starts at an address lower than the one before it.

When the Assembler defines a `DECREMENT` data module, it locates the module's identifier at an entry point outside the module and at a higher address. Thus the actual module is **anonymous**; you cannot access it directly. Further, since the Assembler gives the module's identifier an offset that is equal to the size of the module, the identifier remains undefined until the completion of the module's definition. This means it cannot be accessed inside the module by expressions that require all identifiers to be previously defined, such as equates.

In terms of their actual structure in the object file, all code and data modules containing  $n$  bytes are considered to have their bytes numbered positively from 0 to  $n - 1$ . With code modules and incrementing data modules, their location-counter offsets correspond directly to their object file numbering. With decrementing data modules, however, their location-counter offsets and object file numbering are complementary. Byte 0 in the object file corresponds to the end of the last byte of the last piece of data in the module.

To illustrate this, suppose you defined a data module consisting of three long words:

Location counter	Data	RECORD	, DECREMENT	Object file bytes
				<b>12</b>
-4	a	DS.L	1	<b>8</b>
-8	b	DS.L	1	<b>4</b>
-12	c	DS.L	1	<b>0</b>
		ENDR		

The numbers on the left are the generated offsets as determined by the location counter. The numbers on the right are the module offsets in the object file. Since the Assembler generates references to the module as offsets from the module identifier, the Assembler's negative offsets will work only if we define the identifier as byte  $n$  of the module (not as byte 0), where  $n$  is the size of the module—in the above example, 12. In this way the identifier specifies an entry point in an anonymous module.

You can write INCREMENT as INCR and DECREMENT as DECR.

## MAIN

The parameter value MAIN in a RECORD directive generates a special form of decrementing data module, called the **main data module**. When it collects all the data modules in your source text together, the Linker normally adjusts the A5 offsets in all code statements that access data. This means that the Linker must retrieve all referenced data locations from the object file. By declaring one data module as MAIN, you can shorten this process.

A program can have only one main data module. The Segment Loader loads it first, immediately below A5. Because the position of the main data module is unique, the Assembler can adjust the code statement offsets that access it without relying on the Linker. The Linker, in turn, does not retrieve the locations of data in the main data module and no Linker records are generated for it in the object file. As a result, the unlinked object file is smaller and the Linker runs faster.

Because the main data module is loaded below A5 and its offsets are generated by the Assembler, the generated offsets are negative. Therefore the main data module is always a decrementing data module.

---

## CODE and DATA: Switch between code and data

```
[macro-label]    CODE
[macro-label]    DATA    { [ INCR [ EMENT ] ]
                          { [ DECR [ EMENT ] ]
                          [ MAIN ] }
```

You can define an associated data module during the definition of a code module, without ending the current code module, by using `CODE` and `DATA`. (This technique is illustrated in Chapter 2 under "Source Text Structure.")

`CODE` and `DATA` may be used only inside a code module—a module defined by `PROC`, `FUNC`, or `MAIN`. The `DATA` directive switches the Assembler to defining a data module; `CODE` switches it back to defining the original code module. The final result is one contiguous code module and one contiguous data module, regardless of how many times you use `CODE` and `DATA`. Remember that your source text may contain only directive statements when `DATA` is in force; it may contain both machine instruction statements and directive statements when `CODE` is in force.

The `DATA` directive can generate either an incrementing, decrementing, or main data module, depending on the value of its parameter. With no parameter, it generates an incrementing data module. A full explanation of these options is given under "RECORD and ENDR" earlier in this chapter. The option you select the first time you use `DATA` in a given code module governs all data generation within that module; the Assembler ignores subsequent `DATA` parameters until the code module ends. You can generate different kinds of data modules from different code modules, however. You can use `MAIN`, with either `RECORD` or `DATA`, only once in a program.

You can write `INCREMENT` as `INCR` and `DECREMENT` as `DECR`.

---

## END: End the assembly

```
[macro-label]    END
```

The `END` directive marks the end of your assembly. The Assembler ignores any source text after `END`.

`END` is a required directive. The Assembler generates a warning (not an error) if it is omitted. You must not place `END` in a file called by an `INCLUDE` directive, unless you intentionally want to terminate your assembly from the included file.

---

## Symbol definitions

The directives described in this section let you assign values to individual identifiers. They let you name certain objects—numeric constants, individual registers, register lists, and opcodes—so that you can use the identifiers instead of the original objects in your source text. The directives are as follows:

EQU	Assign a permanent value to a symbol
SET	Assign a temporary value to a symbol
REG	Assign an identifier to a processor register list
FREG	Assign an identifier to an MC68881 register list
OPWORD	Assign an identifier to an opcode

---

### EQU and SET: Name constants and registers

<i>name</i>	EQU	$\left. \begin{array}{l} \textit{arith-expr} \\ \textit{reg} \\ \textit{import-name} \end{array} \right\}$
-------------	-----	--

<i>name</i>	SET	$\left. \begin{array}{l} \textit{arith-expr} \\ \textit{reg} \\ \textit{import-name} \end{array} \right\}$
-------------	-----	--

EQU and SET assign the value in the operand field to the identifier in the label field. Both fields are required. These directives are collectively called **equates**. The operand may be a numeric expression, a register name, or an identifier imported from another module.

EQU assigns a permanent value; once an identifier has been used in an EQU directive it may not be redefined in another EQU directive within its scope, with the one exception that an EQU with the same value generates a warning. SET assigns a temporary value; the same identifier may be redefined with another SET directive.

When you use EQU or SET with a numeric expression, follow these rules:

- The numeric expression *arith-expr* must not contain any forward or undefined references.
- Relocatable expressions are allowed only inside code modules and data modules.
- Equates defined outside modules or inside code modules may take any value. If you use an equate in a template or data module, its value must be in the range -32768..+32767 (that is, a signed 16-bit value).

You can use `EQU` or `SET` with any of the register names listed in Table 3-4 or with any identifier previously equated to one of those register names.

You can equate an identifier to a floating-point constant or to any identifier previously equated to a floating-point constant, but only if the `MC68881` directive is in effect. Because such constants are not evaluated until used (by a `DC` directive or an `MC68881` machine instruction), `EQU` and `SET` store their values as strings and do not validate them.

Equates to absolute values (constants and registers) must appear in your source text before you use the equated identifiers. When the Assembler encounters a symbol in an effective address in a code module, it searches for its value first in the code module's local symbol table (if the symbol has been defined), then in the global symbol table. This means that effective addresses may not contain forward references to equates defining absolute values or registers. Forward references to relocatable equated values (for example, equates to the location-counter value) are allowed, since the Assembler always assumes that forward references refer to relocatable objects.

**▲ Warning** If you give the same identifier to a forward-referenced local label as you give to a global absolute equate symbol, the Assembler uses the value of the global symbol and issues a name conflict warning. This occurs because the local identifier is not yet defined. Here is an example:

```
Piotrus      EQU 7
Alek  PROC
            MOVE #Piotrus,A2
Piotrus      MOVE #0,A1
            END

### Warning 233 ### Possible name conflict with
global symbol: PIOTRUS File "hd40:MPW:Worksheet";
line 4 ▲
```

Here are some examples of valid equates:

```
Length      EQU      *-Start      ; Define Length,
                                           ; Start to location counter
Cr          EQU      $0D           ; Define the return character
X          SET      Y+10          ; Define X as the value of Y+10
X          SET      Y+20          ; Redefine X as the value of Y+20
StkPtr     EQU      A7           ; Define StkPtr as register A7
ProgCtr    EQU      PC           ; Define ProgCtr as
                                           ; the program counter
SuppA2     SET      ZA2          ; Define SuppA2 as a
                                           ; zero-suppressed A2
```

```
Alpha      EQU      (a+b)*10      ; Define Alpha with the
; expression's value
Pi         EQU      "3.14159"    ; Define Pi as a floating-point
; constant
```

---

## REG and FREG: Name register list

```
name      REG      rlist
name      FREG     fp-rlist
```

The REG and FREG directives assign the register list *rlist* or *fp-rlist* to the specified *name*. Lists named by REG are used with MOVEM instructions; lists named by FREG are used with FMOVEM instructions. You can use FREG only if the MC68881 directive is in force. Simple register lists are composed as follows:

- *Rm-Rn* designates registers *Rm* through *Rn* (where  $m \leq n$ , and *Rm* and *Rn* are both A registers or both D registers).
- *Ri/Rj/Rk...* designates registers *Ri*, *Rj*, *Rk...* where each term is an A register, a D register, or a range *Rm..Rn*.
- *FPm-FPn* designates floating-point registers *FPm* through *FPn* ( $m \leq n$ ).
- *FPI/FPj/FPk...* designates registers *FPi*, *FPj*, *FPk...*. Each term is either a single register *FPn* or a range *FPm..FPn*.
- *FPCR/FPSR/FPIAR* designates the floating-point control registers *FPCR*, *FPSR*, and *FPIAR*, in any order. You cannot combine these registers with other registers in a register list.

Here are some examples:

Example	Meaning
D0-D1/A3	D0, D1, and A3
D2-D4/A1-A2/D7	D2, D3, D4, A1, A2, and D7
FP0-FP3/FP7	FP0, FP1, FP2, FP3, and FP7
FPCR/FPSR	Control registers FPCR and FPSR

The scope and search rules for register-list identifiers are exactly the same as for equate identifiers, as discussed earlier under "EQU and SET."

You can use identifiers defined by REG and FREG to build up more complex register lists. To do this, you concatenate them with register lists or other register-list identifiers, as shown here:

```
VolatileDs  REG      D0-D2          ; Volatile D registers
VolatileAs  REG      A0-A1          ; Volatile A registers
VolatileRegs REG     VolatileAs/VolatileDs ; Volatile A and D registers
ActiveRegs  REG      VolatileRegs/D6-D7/A4 ; All required registers
```

In this example, the register list `ActiveRegs` is defined so that it is equivalent to the simple list `D0-D2/D6-D7/A0-A1/A4`.

Here is a sample program fragment that shows `REG` and `FREG` directives used with `MOVEM` and `FMOVEM` statements:

```

PascalRegs  REG      D2-D7/A3-A5      ; Names Pascal registers
FPRegs      FREG     FP0-FP7      ; Names FP registers
- - -
P           PROC      EXPORT
           LINK      A6,#-LocalSize
           MOVEM.L   PascalRegs,-(A7) ; Save Pascal registers
           FMOVEM.X  FPRegs,-(A7)    ; Save FP registers
           - - -
           FMOVEM.X  (A7)+,FPRegs    ; Restore FP registers
           MOVEM.L   (A7)+,PascalRegs ; Restore Pascal registers
           RTS
           ENDPROC

```

---

#### OPWORD: Name machine instruction

*name*            OPWORD            *abs-expr*

OPWORD is used to assign a numeric value to the identifier *name* so that it may subsequently be used as a machine instruction. The expression *abs-expr* must have an absolute value in the range 0..65535 (`$0..$FFFF`, hexadecimal) and may not contain any forward, undefined, or imported references. Identifiers defined by OPWORD may be used only inside code modules.

When the Assembler processes any mnemonic, it searches the following lists in the order shown:

1. standard opcodes and directives, including coprocessor instructions
2. macro identifiers
3. OPWORD names in the code module's local symbol table
4. OPWORD names in the global symbol table

Although the Assembler makes no assumptions about the use of OPWORD definitions, the intent of OPWORD is to allow you to define the Macintosh trap values. For example, you could define `_Read` as an identifier for the Macintosh File Manager read trap (`$A002`) as follows:

```

_Read      OPWORD    $A002      ; Define read trap call

```

To generate the value represented by an OPWORD name, use the name just like an Assembler mnemonic or macro call. For example, after the OPWORD directive just illustrated, the following causes the Assembler to generate an opcode of value \$A002:

```
    _Read                ; Generate $A002 trap call
```

Names defined by OPWORD may be used with parameters. The general syntax for the use of an OPWORD name is

```
[[label | macro-label]]      opword-name      [abs-expr],...
```

The expressions *abs-expr* must have absolute values in the range 0..65535 (\$0..\$FFFF) and must not contain any forward, undefined, or imported references. Each value is combined under the rules governing logical OR with the value of *opword-name* to produce the final generated machine instruction code.

Hence the earlier example could be extended by means of the following equate:

```
Async    EQU            $400        ; Defines "async" bit for
                                           ; File Manager traps
```

The original `_Read` statement with a parameter would then generate \$A402:

```
    _Read    Async      ; Generates $A402 trap call
```

OPWORD parameters must be separated by commas, but there need not be any expressions between commas. Two adjacent commas delimit an expression that does not affect the generated instruction. Hence the following statement also generates \$A402:

```
    _Read    ,Async,,, ; Generates $A402
```

- ◆ *Note:* The standard Macintosh trap macros, discussed under "Macintosh Libraries" in Chapter 1, consist largely of OPWORD directive statements.

---

## Data definitions

The data-definition and storage-allocation directives described in this section let you define constants, initialize data, and reserve storage areas in code modules, data modules, and templates. They are the following:

```
DC      Place constants in a code or data module
DCB     Place a block of constants in a code or data module
DS      Define a storage area
```



---

## DC and DCB: Place constants in code or data

```
{ { label | macro-label } }      DC[.size]      { expr | string },...  
{ { label | macro-label } }      DCB[.size]     length, { expr | string }
```

DC and DCB place data in the current (code or data) module. When used outside an existing module, they define a new data module containing the specified data. The optional qualifier *size*, which is separated from the directive name by a period, consists of a letter that indicates the size of each data increment. Word (w) is the default value if you do not include the qualifier. *Size* also determines the size of the increments specified by the integer expression *length*, as shown in Table 4-1.

■ **Table 4-1** DC and DCB data increments

Qualifier	Name	Length increments, in bytes
B	Byte	1
W	Word	2
L	Long word	4
S	Single precision	4
D	Double precision	8
X	Extended	12
P	Packed BCD	12

The operand field of a DC directive statement may contain up to 25 values, numeric expressions, and strings in any mixture, separated by commas.

The operand field of a DCB directive statement begins with a length expression that specifies the number of data increments in the data block. The size of each increment is determined by the *size* qualifier, as shown in Table 4-1. This is followed by a single value to be placed in each such increment. Hence a DCB directive statement with a *length* of *n* acts the same as *n* DC directives. The DCB *length* parameter must be an absolute expression with a value greater than 0, and may not contain any forward, undefined, or imported references.

All the values specified in a single DC or DCB directive statement make up one data module if the statement is used outside a code or data module. All the values make up one block of ascending bytes if it is used inside a code or data module. This is true even when the data module is declared as having a decrementing location counter.

All data sizes except byte (B) are aligned to the next word boundary unless an ALIGN 0 directive is in force. The optional *label* is associated with the first byte of data after alignment.

Integer expressions must fit into the size specified by the DC or DCB size qualifier. For example, a value of 1000 cannot be used with a DC . B directive. Strings are formatted according to the current STRING directive setting. When used with a string value, the DC or DCB size qualifier affects alignment only.

Here are some examples of size qualifiers and data values:

```

DC.B      'Nebur L. Ari'      ; A 12-character string in
                                ; current format
DC.L      1, 2, 3             ; Three long words containing 1, 2,
                                ; and 3
DC.B      T1-T2               ; A byte with two relocatable
                                ; references
DC.B      $FDF                ; An error ($FDF is too big for a byte)
DC.W      1                   ; A word constant containing integer 1
DC.X      "1.234"             ; A 12-byte extended constant
DC.D      "Nan(1)"            ; An 8-byte double-precision constant
DC.L      '1234'              ; A 4-character string in current
                                ; format
DC.L      ('1234')            ; A 4-byte constant $31323334

```

The last example is a four-character string enclosed in parentheses. Because both strings and integer expressions may be used as DC or DCB operands, the Assembler decides the operand's type by examining its first symbol. The parentheses force the Assembler to type the operand as an integer expression. As such, it can contain a string constant of up to four characters without exceeding the long-word size set by the L qualifier; the Assembler treats it as a right-justified 32-bit value padded on the left with zeros. In the next-to-last example, the Assembler treats the operand as an ordinary string constant.

You must take care when using DC and DCB with imported data parameters. If the current DATAREFS setting is ABSOLUTE (the default value), then any imported data reference is treated as a 32-bit absolute address, requiring a qualifier of L. Other situations require different qualifiers. For further information see "Linker and Scope Controls," later in this chapter.

When you use DC or DCB to place data in a data module, you must link your finished program with the library file Runtime.o, which contains the data initialization routine `_DataInit`. If your main code module is written in assembly language, its first executable statement must be a call (JSR) to the entry point `_DataInit`. This entry point must also be declared as IMPORT. After returning from `_DataInit`, your program may unload the segment `%A5Init` that contains it, by calling the Macintosh routine `UnloadSeg`. If your main program is written in C or Pascal, no explicit call to `_DataInit` is required, because the run-time libraries for C and Pascal automatically take care of data initialization.

---

## DS: Define storage area

```
{ { label | macro-label } } DS [ size ] { length }  
{ template-name }
```

The DS directive allocates and defines an uninitialized storage area in a code module, data module, or template. When used outside an existing module, it defines a new data module of the specified length. The optional qualifier *size*, which is separated from the directive name by a period, consists of a letter that indicates the size of each of the data increments defined by *length*, as shown in Table 4-1. Word (w) is the default value if you do not include the qualifier. *Length* must be an absolute expression with a value greater than or equal to zero. It cannot contain any forward, undefined, or imported references.

All data sizes except byte (B) are aligned to the next word boundary unless an ALIGN 0 directive is in force. The optional label is associated with the first byte of data after alignment.

A DS directive with a *length* of 0 aligns code or data to a word boundary. In this form, it ignores any prior ALIGN 0 directive. ALIGN 0 is discussed later in this chapter under "Location-Counter Controls."

The storage area allocated by DS can also be specified by a template identifier that has been previously defined. Template definitions are discussed in the next section. In this case, the length allocated is determined by the size of the template. If you use a *label* with DS and a template identifier, that *label* is given the type represented by the template. You can then access the fields of the template by qualifying its field identifiers with the DS *label* instead of the template identifier, using the form *DSlabel.fieldname*. You can also use DS in the same way to type fields of templates and then access fields within them, using the form *DSlabel.fieldname.innerfield*. You can create nested fields in this way to any depth.

You can use template identifiers to specify DS data types in all cases except when DS is used in the code section of a code module or when there is no label specified. Although the data area allocated by DS is not typed in these cases, its size is still determined by the template's size. When you use a template identifier to specify size or type in a DS directive statement, that identifier must be the directive's only operand. Using the identifier any other way (such as by enclosing it in parentheses or including it in an expression) refers to the identifier's offset value instead of to its type and size.

---

## Template definitions

A **template** describes the layout of a collection of data without actually allocating any memory space. This section describes the following template definition directives:

RECORD	Begin a record template definition
ENDR	End a record template definition
WITH	Begin default record identifier qualification
ENDWITH	End default record identifier qualification

A template definition starts with a RECORD directive statement and ends with an ENDR directive. In between are directives that describe the layout of the template, using DS, ORG, ALIGN, EQU, and SET. Sections of data within a template are called **fields**. Fields are referenced by the form *record.field*, where *record* is the template identifier and *field* is the label in the directive that defined the field. As a convenience, you may use the WITH and ENDMETHOD directives to specify a template identifier over a section of your source text, so you only have to specify the field name. This is like the Pascal WITH statement.

---

### RECORD and ENDR: Define a template

<i>name</i>	RECORD	$\left\{ \begin{array}{l} \textit{offset} \\ \text{IMPORT} \\ \text{'origin'} \end{array} \right\} \left[ \left[ \begin{array}{l} \text{INCR}[\text{EMENT}] \\ \text{DECR}[\text{EMENT}] \end{array} \right] \right]$
		<i>DS, ORG, ALIGN, EQU, and SET directive statements</i>
[ <i>macro-label</i> ]	ENDR	

RECORD and ENDR delimit the section of source text in which you define a template. Notice that RECORD and ENDR are also used to define data modules, as described in "Code and Data Module Definitions," earlier in this chapter. The Assembler distinguishes the two usages by the parameters in the RECORD directive statement. When used to define a data module, RECORD has either no parameters or one of the terminal symbols EXPORT or ENTRY as its first parameter. When used to define a template, RECORD always has at least one parameter—the absolute expression *offset*, the terminal symbol IMPORT, or the identifier *origin* enclosed in braces.

The definition of a template is equivalent to a sequence of equates. However, it is a more natural way to specify a storage layout. Templates may be defined only outside modules or as local definitions inside code modules.

As with code and data modules, templates have their own location counters corresponding to the next available data location. As each data field is defined, the location counter is incremented by the size of that data field. The next piece of data is then placed at the next available location. Location-counter values must be in the range -32768..+32767.

To define field locations in both positive and negative directions, you can include `INCREMENT` or `DECREMENT` in the `RECORD` directive, preceded by a comma. `INCREMENT` is the default parameter; it makes `RECORD` allocate fields at ascending locations, as just described. If you specify `DECREMENT`, fields are located at descending locations, corresponding to Pascal memory layouts. Before defining each field, the Assembler decrements the location counter by its size; hence each field starts at an address lower than the one before it.

The parameter *offset* represents an initial offset for the template. It must be an absolute expression without any forward, undefined, or imported references. Specifying a nonzero offset is equivalent to specifying a zero offset and placing an `ORG` directive at the start of the template definition, as shown in these examples:

Name	RECORD	100		Name	RECORD	0
			defines the same template as		ORG	++100
	- - -				- - -	
	ENDR				ENDR	

The main advantage of the specification on the left is that the template name takes the value of the initial offset. The template name can then be used in place of the offset value in arithmetic expressions.

You can specify a negative offset with `RECORD`. This is useful for mapping Macintosh Pascal stack frames. Suppose, for example, you want to write an external Pascal procedure `Px` with the following declaration:

```
PROCEDURE Px(a,b,c: INTEGER); EXTERNAL;
```

If the Pascal program calls this procedure in the form `Px(a, b, c)`, then the following equivalent code actions are generated by Pascal:

```

MOVE.W    a(A6),-(A7)    ; Push a
MOVE.W    b(A6),-(A7)    ; Push b
MOVE.W    c(A6),-(A7)    ; Push c
JSR       Px             ; Call external procedure Px

```

In assembly-language procedure `Px`, you want to reserve stack space for local variables. So, following the conventions used by the MPW Pascal Compiler, start the subroutine with `LINK A6` to reserve the local stack space, as follows:

```

Px        PROC          EXPORT
          WITH          StackFrame
          LINK          A6,#LocalSize ; Reserve space for locals on stack
          - - -
          RTS
          ENDP

```

You can now define the stack frame, using `RECORD` to delimit the following template definition:

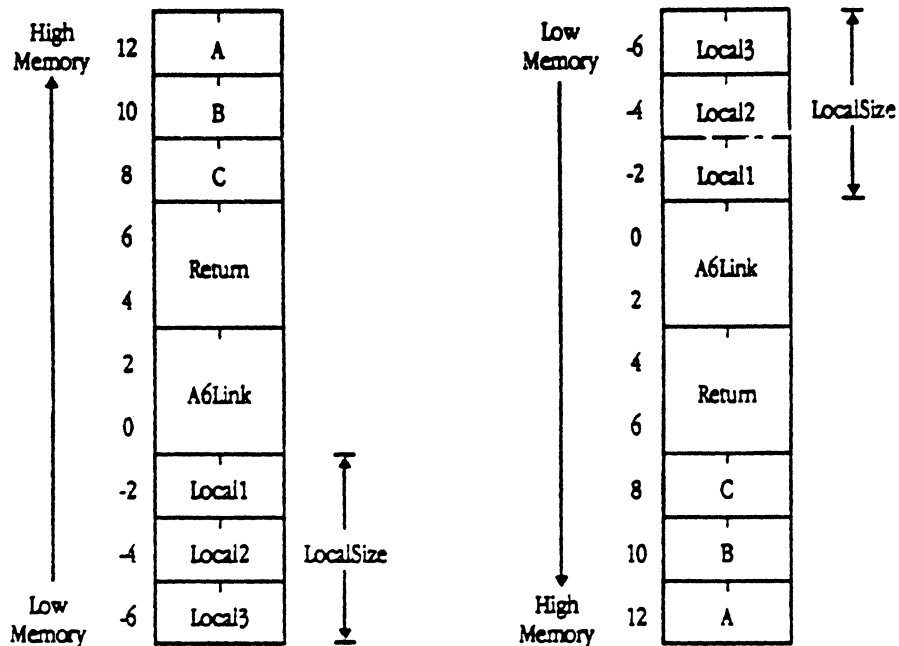
```

StackFrame RECORD      -6      ; Start at -6 for 3 local integers
Local3     DS.W        1        ; Third local
Local2     DS.W        1        ; Second local
Local1     DS.W        1        ; First local
LocalSize  EQU         Local3-* ; Local area (-6) used in LINK
A6Link     DS.L        1        ; Old value of A6 set by LINK
Return     DS.L        1        ; Return address for RTS
C          DS.W        1        ; c parameter
B          DS.W        1        ; b parameter
A          DS.W        1        ; a parameter
          ENDR

```

Figure 4-1 shows the stack frame after the `LINK A6` instruction in the example has been executed. It illustrates two ways to view the same template layout.

■ Figure 4-1 Stack frame example



The low-to-high layout on the right matches the record template definition just given because the RECORD directive assumed the default parameter INCREMENT, even though it contained a negative initial offset. By specifying DECREMENT, you could equally well define the template to match the high-to-low diagram on the left:

```

StackFrame      RECORD      14,DECR      ; Start at 14 and decrement
A                DS.W        1            ; a parameter (at location 12)
B                DS.W        1            ; b parameter (at location 10)
C                DS.W        1            ; c parameter (at location 8)
Return          DS.L        1            ; Return addr for RTS
                ; (location 4)
A6Link          DS.L        1            ; Old A6 value set by LINK
                ; (location 0)
Local1          DS.W        1            ; First local (at location -2)
Local2          DS.W        1            ; Second local (at location -4)
Local3          DS.W        1            ; Third local (at location -6)
LocalSize       EQU         *            ; Local area (-6) used in LINK
                ENDR

```

Notice that in both of the foregoing stack frame layouts the initial offset had to be given. For most mappings the offset will be 0. However, for stack frames the initial offset must be chosen so that the `A6Link` field will have an offset of 0. This is the reason that you specified `-6` in the sample incrementing layout and `14` in the sample decrementing layout. It is not necessary, however, to compute the size of each template and enter it as an absolute value. You can use the `{origin}` parameter to make the Assembler do this work for you. The template origin is defined as the field which is to have an offset of 0. You specify a field identifier enclosed in braces to indicate that that field is to be the template's origin. The Assembler then reads in the the template definition as if the initial offset was 0 (displaying these values in the Assembly listing) and subtracts the zero-relative offset of the origin field from each field offset. The effect is to shift the template's origin from the start of the template to the field specified by the origin parameter.

Hence in the preceding examples, you could have used simpler `RECORD` directive forms, leaving the rest of the template definitions unchanged:

```
StackFrame RECORD (A6Link)
StackFrame RECORD (A6Link),DECR
```

Notice that shifting the origin of a template affects only the field offsets. Equates are not changed. In origin-shifted templates, the Assembler distinguishes between equates to absolute expressions (such as `Loca13-*` in the incrementing example) and equates to other field identifiers.

Normally you define some dynamic data, such as the stack frame illustrated in Figure 4-1, and then use a template to map over the data. However, you may also have static data, defined somewhere else in your assembly-language program as a data module, that you want to map. Because both templates and data modules are defined by `RECORD` directives, they have the same underlying form. This lets you import an entire data module and directly access its fields. You map a data module by specifying its identifier in an `IMPORT` directive and then using that same identifier as a template label in a `RECORD` directive. Alternatively, you can specify the `IMPORT` directive identifier explicitly as the `RECORD` template parameter. The base register for a data module imported as a template is always `A5`.

In C programming, the situation just described corresponds to declaring a static structure (`struct`) as `external`, and then importing the entire structure (`extern`) from another file. The `struct` declaration must appear in both files, just as the `RECORD` directive appears twice in the assembly-language source text.



## Using templates as data types

In higher-level languages, data types define the specific ways that data is stored in memory. For example, a Pascal record type or a C struct type specifies the memory layout and size of all data items of that type. Fields within data structures also have types; hence higher-level languages allow the creation of complex structures of typed data.

In the MPW assembly language, templates serve the same purpose. To create the equivalent of a data type, you use the identifier of a template (the label you used in the RECORD directive that created it) as the operand of a DS directive. Here is an example:

```
label          DS          template-name
```

Specifying a template identifier alone makes the DS directive allocate an amount of memory equal to the size of the template. If a label is also specified, that label acquires the type represented by the specified template. You can then access fields of the template by qualifying the field identifiers with the DS label instead of the template identifier, in the form *label.fieldname*. Fields of templates can themselves be typed the same way. You can identify them by a series of qualifications, in the form *label.fieldname.innername*. Fields can be nested this way to any depth.

The following is an example of how template types are used. The example shows the Macintosh QuickDraw definitions for points and rectangles. The corresponding Pascal type declarations are shown as comments:

```
Point    RECORD 0      Point = RECORD CASE INTEGER OF
v        DS.W  1      0: (v: INTEGER;
h        DS.W  1      h: INTEGER);
        ORG    v
vh       DS.W  2      1: (vh: ARRAY [2] OF INTEGER)
        ENDR      END;

Rect     RECORD 0      Rect = RECORD CASE INTEGER OF
top      DS.W  1      0: (top:    INTEGER;
left     DS.W  1      left:   INTEGER;
bottom  DS.W  1      bottom:  INTEGER;
right   DS.W  1      right:   INTEGER);
        ORG    top
topLeft  DS    Point  1: (topLeft: Point;
botRight DS    Point  botRight: Point)
        ENDR      END;
```

In this example, both `topLeft` and `botRight` are defined as having the type `Point`. `Point` and `Rect` may now be used to allocate space in a data module. For example:

```
MyData   RECORD          ; Define a data module
- - -
MousePt  DS              Point
DragRect DS              Rect
- - -
        ENDR
```

You can now access the various fields of `MousePt` and `DragRect` by using the field identifiers established in the template definition. First, though, you must make these field labels known to the code by bracketing them with a `WITH MyData...ENDWITH` pair as described in the next section. Because these fields are in a data module, the base register is `A5`. Here are some examples:

```

MOVE.W   MousePt.v(A5),D0           ; Get v component
MOVE.L   MousePt.vh(A5),D0         ; Get full point
                                           ; position
MOVE.W   DragRect.left(A5),D0      ; Get left coordinate
MOVE.L   DragRect.topLeft(A5),D0   ; Get topLeft of
                                           ; rectangle
MOVE.W   DragRect.botRight.h(A5),D0 ; Get botRight h
                                           ; component

```

You can use templates as types in `DS` directives any time except when you use `DS` in the code section of a code module, or when the `DS` directive has no label. Although the Assembler does not establish a type in those cases, it still uses the template size to define the space allocated by `DS`. To use a template identifier as a size or type specification, you must supply it as the only operand in the `DS` directive statement. Using the identifier any other way (for instance, enclosing the identifier in parentheses or using the identifier in an expression) makes `DS` use the template's value instead of its identifier.

In the foregoing example, incrementing templates were used to define types. Decrementing templates may also be used. If you use a decrementing template as a type, its origin for data allocation purposes is shifted so that its lowest address corresponds to a location-counter value of 0. You can freely mix incrementing and decrementing templates to define complex data types.

---

### WITH and ENDWITH: Supply RECORD name qualification

```

[macro-label] WITH      name,...
                  Code-module statements
[macro-label] ENDWITH

```

The `WITH` directive lets you access `RECORD` field identifiers without explicit qualification. `WITH` may only be used inside code modules (modules delimited by `PROC`, `FUNC`, or `MAIN`). You can write a series of identifiers, separated by commas, as parameters; they all become field qualifiers. They may be the identifiers of data modules, templates, or typed fields. For a description of field typing see "Using Templates as Data Types," earlier in this chapter. The implicit qualification established by `WITH` remains in effect until a matching `ENDWITH` or the end of the code module.

Using the StackFrame RECORD template illustrated earlier, the following example shows how WITH can be used to access a template's fields:

```

LINK    A6,#StackFrame.LocalSize           WITH    StackFrame
MOVE    StackFrame.A(A6),D0                LINK    A6,#LocalSize
MOVE    StackFrame.B(A6),D1                MOVE    A(A6),D0
MOVE    StackFrame.C(A6),D2                MOVE    B(A6),D1
-- -- --                                  MOVE    C(A6),D2
-- -- --                                  -- -- --
-- -- --                                  ENDWITH

```

WITH directives may be nested. Alternatively, more than one identifier may be specified in a single WITH directive. The latter is equivalent to nesting WITH directives, with the last parameter being considered the most deeply nested:

```

WITH    alpha,beta,gamma                   WITH    alpha
-- -- --                                  is
-- -- --                                  WITH    beta
-- -- --                                  equivalent
-- -- --                                  to
-- -- --                                  WITH    gamma
-- -- --                                  -- -- --
-- -- --                                  ENDWITH
-- -- --                                  ENDWITH
ENDWITH                                    ENDWITH

```

You can nest field qualifications, using WITH directives in either form. Parameters occurring earlier will qualify parameters occurring later. The Assembler searches for each specified field identifier, starting with the most deeply nested WITH, and attaches the qualification when it finds it. If two fields have identical identifiers, it supplies the most deeply nested qualification. Here is an example, based on the DragRect definition given in "Using Templates as Data Types" in the discussion of REC and ENDREC:

```

WITH    DragRect,topLeft ; Qualify with DragRect
-- -- -- ; and DragRect.topLeft
MOVE.W  v(A5),D0         ; DragRect.topLeft.v
MOVE.W  left(A5),D2      ; DragRect.left
MOVE.W  botRight.h(A5),D1 ; DragRect.botRight.h
ENDWITH

```

Notice here that topLeft is subject to the WITH qualification of DragRect, the WITH directive's first parameter. This is equivalent to the explicit qualification DragRect.topLeft. The v field reference is qualified by topLeft, so that it is equivalent to a reference to DragRect.topLeft.v. The left field is a field of DragRect, so that it is equivalent to a reference to DragRect.left. The last reference is to the h field of botRight, which is itself a field of DragRect. However, h also occurs as a field identifier in topLeft. The explicit reference to botRight is required to override the implicit qualification topLeft.

As you can see from the last example, nested `WITH` directives can generate unintended field identifier qualifications, leading to very subtle program bugs. (It would have been simple, and incorrect, to refer to `h` without realizing it was the `topLeft` when you meant the `botRight`.) Use `WITH` only to cover short sections of source text, and avoid complex nestings. If in doubt, replace `WITH` directives with fully qualified field identifiers. Your program will become easier to understand and maintain when the reader always knows to which module or template every field belongs.

---

## Linker and scope controls

The directives described in this section all pass information to the MPW Linker. They tell the Linker how to associate identifiers between object files, how to group individual modules into segments, and how to comment the object-code file. At the same time, they give the Assembler information about the scope of objects named in the directives and tell it whether they are code or data. They are the following:

<code>EXPORT</code>	Make entry points accessible in other assemblies
<code>ENTRY</code>	Make local entry-points global
<code>IMPORT</code>	Identify entry points declared externally
<code>CODEREFS</code>	Control the linking of code-to-code references
<code>DATAREFS</code>	Control the linking of data-to-code and data-to-data references
<code>SEG</code>	Specify the current code segment
<code>COMMENT</code>	Place a comment in the object file

`ENTRY`, `EXPORT`, and `IMPORT` all affect the scope of code or data module identifiers. `ENTRY` promotes an identifier to global scope within a file, so that it is accessible to all references in the same file. `EXPORT` has the same effect as `ENTRY`; in addition, it makes the identifier accessible to other files, or global to the assembly. `IMPORT` provides a reference in the current module or assembly for identifiers exported in another module, assembly, or compilation.

`CODEREFS` and `DATAREFS` allow you to control some of the characteristics of the way the Linker associates code and data references created by `EXPORT`, `ENTRY`, and `IMPORT`.

`SEG` specifies the code modules in a segment. For a discussion of code module segments, see "Segmentation" in Chapter 2.

`COMMENT` tells the Linker to generate a comment record for your object file.

---

## EXPORT and ENTRY: Expand scope of entry points

[macro-label]      ENTRY       $\left\{ \begin{array}{l} (name_1, name_2, \dots) : \begin{cases} \text{CODE} \\ \text{DATA} \end{cases} \\ name_1 : \begin{cases} \text{CODE} \\ \text{DATA} \\ \text{MAIN} \end{cases}, \dots \end{array} \right\}, \dots$

[macro-label]      ENTRY       $\left\{ \begin{array}{l} (name_1, name_2, \dots) : \begin{cases} \text{CODE} \\ \text{DATA} \end{cases} \\ name_1 : \begin{cases} \text{CODE} \\ \text{DATA} \\ \text{MAIN} \end{cases}, \dots \end{array} \right\}, \dots$

With the exception of local labels, all identifiers in a code or data module are automatically accessible throughout the module in which they are defined. **EXPORT** and **ENTRY** extend the scope of specified identifiers by making them accessible in other modules as well. **EXPORT** makes them accessible in modules in all files linked with the file containing it; **ENTRY** makes them accessible only in modules within the same assembly.

Identifiers listed with **EXPORT** are said to be *exported*. Each one must be designated as code or data; one may be designated as **MAIN**. You can accept the default designations or you can specify them explicitly. These rules govern how the Assembler treats the operands of **EXPORT** and **ENTRY** directives by default:

- The default designation for identifiers listed inside a code module is **CODE**.
- The default designation for identifiers listed inside a data module is **DATA**.
- The default designation for identifiers listed outside any module is **CODE**.

You can override any of these default designations by writing explicit declarations. If you specify **CODE**, **DATA**, or **MAIN** explicitly, you can write a series of identifiers separated by commas and enclosed in parentheses, followed by a declaration:

```
EXPORT      (name1, name2, name3): DATA
```

Alternatively, you can write a separate declaration for each name, omitting the parentheses:

```
ENTRY  
name1: DATA, name2: CODE, name3: DATA
```

Here are the rules for using EXPORT and ENTRY:

- You must place each EXPORT or ENTRY directive in your source text before defining any of the identifiers it affects.
- The directive must be written within the existing scope of all identifiers it affects.
- An exported identifier may not be identical to any other identifier within its new scope.
- You export a module identifier either by using EXPORT or ENTRY before the directive that starts the module (PROC, FUNC, MAIN, or RECORD) or by including EXPORT or ENTRY in the module directive's parameter list.
- You export identifiers occurring within a module by including an EXPORT or ENTRY directive inside the module, before they are defined.
- An identifier may be mentioned in more than one EXPORT or ENTRY directive, provided it is not listed as both code and data.
- An EXPORT directive mentioning an identifier previously listed in an ENTRY directive supersedes the ENTRY.
- An ENTRY directive mentioning an identifier previously listed in an EXPORT directive has no effect.
- An identifier designated as CODE in an ENTRY or EXPORT directive may be later designated as MAIN by an ENTRY, EXPORT, or MAIN directive.
- Only one identifier in an assembly may be designated as MAIN.
- You cannot include a qualification when exporting a field identifier. Only the unqualified field identifier will be exported.

The following example illustrates the placement of EXPORT statements in a file:

```

      EXPORT      X                ; Export code module name X
      EXPORT      Y:DATA           ; Export data module name Y
Y     RECORD      ; Could have exported from here
      EXPORT      Field1,Field2    ; Export inside module
Field1 DS.W       1
Field2 DS.L       2
      ENDR
X     PROC        ; Could have exported from here
      EXPORT      Z                ; Declare secondary entry point
      - - -                    ; Code for X
Z     ; Secondary entry point Z
      - - -                    ; More code for X
      END
```

---

## IMPORT: Identify external entry points

[ *macro-label* ]      **IMPORT**       $\left\{ \begin{array}{l} (name_1, name_2, \dots) : \left\{ \begin{array}{l} \text{CODE} \\ \text{DATA} \\ \text{type} \end{array} \right\} \\ \\ name_1 : \left[ \begin{array}{l} \text{CODE} \\ \text{DATA} \\ \text{type} \end{array} \right], \dots \end{array} \right\}, \dots$

The **IMPORT** directive makes specified identifiers accessible to the file or module in which it occurs. Such identifiers are said to be "imported." Every imported identifier must be declared elsewhere in one of the following ways:

- as either **ENTRY** or **EXPORT** in other modules of the same assembly
- as **EXPORT** in other assemblies
- as an exported procedure, function, or global variable in another language

In the syntax diagram given here, *type* is the identifier of a template used to define a record structure, as explained in the discussion of **RECORD** and **ENDR** in "Template Definitions," earlier in this chapter. As explained there, the template itself may be declared as **IMPORT**. Using an **IMPORT** directive, however, lets you import several templates under other identifiers without having to modify the original template declarations.

Here are the principal rules governing the use of the **IMPORT** directive:

- The inclusion of an identifier in an **IMPORT** statement is treated as a definition of the identifier with respect to identifier scope. This means that imported identifiers follow the standard local/global scope rules covered in "Scope of Definitions" in Chapter 2.
- Imported identifiers that are to be made accessible to more than one module in a file must be imported before any modules that use the identifiers are defined.
- Imported identifiers local to a module must be listed in an **IMPORT** directive inside that module before they are mentioned in any other statement.
- The Assembler does not verify the **CODE** or **DATA** designation of imported identifiers. Hence you should give them the same designation they had when they were exported.
- You can access fields of imported templates by qualifying the identifiers used in **IMPORT** with the original field identifiers.
- Imported code identifiers may be used in all PC-relative effective address modes. However, their use in short branches and in indexed modes with 8-bit displacements may result in run-time errors.

The Assembler gives default CODE or DATA assignments to the operands of the IMPORT directive according to these rules:

- The default designation for identifiers listed inside a code module is code.
- The default designation for identifiers listed inside a data module is data.
- The default designation for identifiers listed outside any module is code.

You can override any of these default designations by writing CODE or DATA explicitly, as described in "EXPORT and ENTRY," earlier in this chapter.

Referring to the example given in "EXPORT and ENTRY," the following example shows how the identifiers exported there could be imported into another file. It also illustrates the identifier scope rules:

```

                IMPORT      X                ; Import X as a code identifier
                IMPORT      (Field1,Y):DATA ; Import Field1 and Y as data

W              PROC
                IMPORT      (Z,L1):CODE     ; Allow local access to Z and L1
                IMPORT      Field2:DATA     ; Allow local access to Field2
                - - -
                MOVE.L      Field2,D1      ; Field2 accessible only
                                           ; from module W
                JSR         L1              ; Call L1 in another module
                MOVE.W      D0,Field1(A5)  ; Copy D0 into Field1
                                           ; (in other file)

                JSR         X
                - - -
                ENDPROC

```

---

## CODEREFS and DATAREFS: Control name linking

```

[macro-label]  CODEREFS      { F[ORCE[JT]]
                             { NOF[ORCE[JT]]
                             { F[ORCE]PC

```

```

[macro-label]  DATAREFS     { R[EL[ATIVE]]
                             { A[BS[OLUTE]]

```

CODEREFS lets you control how the MPW Linker treats code-to-code identifier references—that is, references from one code module to another. With NOFORCEJT, a reference to an address in the same segment will cause the Linker to treat it as PC-relative. References between segments will go through the jump table. With FORCEJT,



all references will go through the jump table even if they are in the same segment. `FORCEPC` is the inverse of `FORCEJT`; it requires that all code-to-code references be PC-relative and in the same segment, and causes a Linker error if any are not. `CODEREFS NOFORCEJT` is the preset condition.

`DATAREFS` lets you control how the Linker treats data-to-code and data-to-data identifier references. With `ABSOLUTE`, all references to code or data are 32-bit absolute jump-table addresses and may be used only in `DC .L` statements. With `RELATIVE`, all references to code are A5-relative jump-table offsets and all references to data are A5-relative offsets. You may use `DC .W` and `DC .L` for `RELATIVE` references. `DATAREFS ABSOLUTE` is the preset condition.

Code-to-data identifier references are always A5-relative; they are unaffected by `CODEREFS` or `DATAREFS`.

You can write operands for `CODEREFS` and `DATAREFS` in any of the following alternate forms:

- `FORCEJT 2S FORCE OF F`
- `NOFORCEJT 2S NOFORCE OF NOF`
- `FORCEPC 2S FPC`
- `RELATIVE 2S REL OF R`
- `ABSOLUTE 2S ABS OF A`

To understand the operation of `CODEREFS` and `DATAREFS` fully, you must know their effects on the linking process. The four possible identifier reference combinations between code and data modules are discussed next in "Code-to-Code References."

### Code-to-code references

When a reference points from one code module to another and `CODEREFS NOFORCEJT` is in effect, the Linker checks to see whether both code modules belong to the same segment. If so, it changes the addressing mode to PC-relative with a 16-bit displacement and sets the appropriate displacement. If the reference is to a code location in a different segment, the Linker converts the address to a location in the jump table (a positive offset from A5). The Linker assumes that the word immediately before the 16-bit displacement represents an instruction which has its destination mode and register fields in bits 0 through 5 (the 6 low-order bits).

- ▲ **Warning**      The Linker does not support editing of the new addressing modes with 32-bit displacements found in the MC68020/MC68030. ▲

The Linker accepts all code references from one module to another. With instructions other than `JSR`, `JMP`, `PEA`, and `LEA`, however, the referenced identifiers must be in the same segment.

To summarize, the Linker follows these rules when checking for illegal code-to-code references:

- If the reference is to a module in the same segment, the Linker accepts it for any instruction and any `CODEREF`s setting. If the instruction being used is not `JSR`, `JMP`, `PEA`, or `LEA`, the referenced identifier must be in the same segment.
- If the reference is to a module in another segment, the Linker accepts it only for `JSR`, `JMP`, `PEA`, and `LEA` instructions, and only if `CODEREF`s `FORCEPC` is not in effect.
- If the reference is to a module in another segment and `CODEREF`s `FORCEPC` is in effect, the Linker will not accept it.

You can use `CODEREF`s `FORCEJT` to forestall certain run-time problems. If, for example, you want to save the PC-relative address of a procedure to call it later and that procedure belongs to a segment that may become unloaded, then attempting to call the procedure after the segment has been unloaded will not work. The solution is to use `CODEREF`s `FORCEJT`. It forces all code-to-code references to go through the jump table anyway, as if the modules were in different segments. Saving a jump table address and using it to call a procedure later guarantees that the corresponding segment will be loaded, even if it is currently unloaded.

### Code-to-data references

When a reference points from a code module to a data module, the Assembler always generates an `A5` offset. If you do not need indexing and the Assembler knows the reference is to data for a machine instruction, you can omit the `A5` reference; the Assembler will generate it for you.

### Data-to-code references

You can reference a code address from a data module only with a `DC` instruction. When you do this, you can choose to make the Linker generate a jump table offset or let it generate the actual run-time jump-table address. If you make the Linker generate an offset, you can specify its size as a word or a long word by qualifying the `DC` instruction (`DC.W` or `DC.L`). If you let the Linker generate the actual run-time jump-table address, you must specify `DC.L`, because a 32-bit address will be added to the `DC` statement at load time. But any `DC` reference to a local label in the same code module (or in a nested data module) will always be treated as a module offset.

DATAREFS controls the two forms of data-to-code addressing. DATAREFS RELATIVE indicates that offsets are to be used, while DATAREFS ABSOLUTE (or no directive at all) indicates that A5-relative 32-bit absolute jump-table addresses are to be generated.

### Data-to-data references

Data-to-data references are similar to data-to-code references. You can force the Linker to generate an A5-relative offset to the data by using DATAREFS RELATIVE, or you can let it refer to the 32-bit absolute address created at run time by doing nothing or by using DATAREFS ABSOLUTE.

Table 4-2 summarizes the effects of CODEREFS and DATAREFS in the four cases just discussed.

- ◆ *Note:* If your program uses absolute data references, you must link it with the library file Runtime.o, which contains the data initialization routine `_DataInit`. If your main code module is written in assembly language, its first executable statement must be a call (JSR) to the entry point `_DataInit`. This entry point must also be declared as IMPORT. After returning from `_DataInit`, your program may unload the segment `%ASInit` that contains it, by calling the Macintosh routine `UnloadSeg`. If your main program is written in C or Pascal, no explicit call to `_DataInit` is required, because the run-time libraries for C and Pascal automatically take care of data initialization. Also, in order to use `UnloadSeg`, you must INCLUDE 'traps.a'.

■ **Table 4-2** Effects of CODEREFS and DATAREFS

From	To	Directive	Effect
Code	Code	CODEREFS FORCEJT	Always uses jump table
Code	Code	CODEREFS FORCEPC	Always PC-relative
Code	Code	CODEREFS NOFORCEJT	Uses jump table if across segments
Code	Data		Always generates A5 offset
Data	Code	DATAREFS RELATIVE	Uses jump table (w or L) offset
Data	Code	DATAREFS ABSOLUTE	Uses 32-bit absolute jump-table addresses (L)
Data	Data	DATAREFS RELATIVE	Generates A5 (w or L) offset
Data	Data	DATAREFS ABSOLUTE	Uses 32-bit absolute addresses (L)

---

## SEG: Specify current code segment

[*macro-label*]      SEG                      [*str-expr*]

All code modules are grouped into segments, as discussed in Chapter 2 under "Source Text Structure." The SEG directive lets you control this grouping. It declares that all subsequent code modules (ignoring any data modules) are to be placed in the segment named by the expression *str-expr*. SEG takes effect at the next PROC, FUNC, or MAIN directive. It remains in effect until the next SEG directive. The modules thus placed in one segment need not be contiguous in the source text.

The default value of *str-expr* is `main` (uppercase *M*, the rest lowercase, as shown). If you do not use the SEG directive or use it without an operand, all subsequent code modules will be placed in the `main` segment.

- ◆ *Note:* Code segment names are case-sensitive. Be careful to use identical capitalization when writing segment names that are to be treated as identical.

Code modules in the same segment do not have to be contiguous in the source file. Code modules belonging to other segments may be mixed with them as long as they fall under the appropriate SEG directive. Here's an example:

```
A            SEG            'Name1 '  
            PROC  
            - - -  
            ENDP  
            SEG            'Name2 '  
B            PROC  
            - - -  
            ENDP  
C            PROC  
            - - -  
            ENDP  
D            SEG            'Name1 '  
            PROC  
            - - -  
            ENDP  
            SEG            'Name2 '  
E            PROC  
            - - -  
            ENDP
```

In this example, modules A and D belong to the segment 'Name1'; B, C, and E belong to the segment 'Name2'.

---

**COMMENT: Place a comment in object file**

*[macro-label]*    COMMENT    *str-expr*

The COMMENT directive lets you place a comment in your unlinked object file. It causes the Assembler to generate an object file comment record containing the value of the COMMENT directive's string expression operand.

---

**Assembly options**

The assembly option directives described in this section let you control certain assumptions the Assembler makes about the program it is assembling. The assembly option directives and the assumptions they control are as follows:

MACHINE	Identify the target microprocessor model
MC68881	Control the assembly of floating-point coprocessor instructions
MC68851	Control the assembly of PMMU coprocessor instructions
STRING	Control the encoding of string constants
BRANCH	Control the encoding of branch instructions
FORWARD	Control the encoding of forward references
OPT	Control the level of code optimization
CASE	Control the treatment of lowercase letters in identifiers
BLANKS	Control the treatment of spaces and tabs in the operand field

---

**MACHINE: Specify target machine**

*[macro-label]*    MACHINE     $\left. \begin{array}{l} \text{MC 68000} \\ \text{MC 68010} \\ \text{MC 68020} \\ \text{MC 68030} \end{array} \right\}$

This directive tells the Assembler that the target microprocessor is an MC68000 (the preset assumption), an MC68010, an MC68020, or an MC68030. The Assembler will accept only those instructions and addressing forms supported by the target microprocessor. A MACHINE directive has effect until the end of the source text file or until another MACHINE directive is encountered.

---

## MC68881: Assemble MC68881/MC68882 coprocessor instructions

`[macro-label] MC68881 [fp-option],...`

This directive tells the Assembler that subsequent source text may contain instructions to an MC68881 or MC68882 Floating-Point Coprocessor and specifies how the Assembler is to interpret them.

Your source text must contain this directive before the first MC68881/MC68882 instruction. Each *fp-option* operand consists of a keyword, an equal sign, and an expression. There are four possible options:

```
C0ID=expr
PREC[ISION]={ X | D | S }
ROUND[ING]={ N | U | D | Z }
KFACTOR=expr
```

The numeric expression *expr* following C0ID is the coprocessor ID number of the MC68881 coprocessor, in the range 1..7. It has a default value of 1.

The letter following PRECISION indicates how much precision and range the Assembler should retain when converting floating-point constants in the source code into binary values. The default value is extended (x); however, you may alternatively specify double precision (D) or single precision (s).

The letter following ROUNDING indicates how the Assembler should round floating-point constants in the source code when converting them into binary values. The default value is to round to the nearest representation (N); however, you may alternatively specify rounding upward (U), downward (D), or toward zero (Z).

The numeric expression *expr* following KFACTOR specifies the default k-factor that the coprocessor uses when interpreting FMOVE . P instructions in which the k-factor is not explicit. The k-factor tells the MC68881 coprocessor in what format to construct the resulting decimal string. The preset default value is -16, but you can specify any value in the range -64..+63. For an explanation of k-factors, see the discussion of FMOVE in the *Motorola MC68881 Floating-Point Coprocessor User's Manual*.

- ◆ *Note:* It is advisable to program for the MC68882 even if your target hardware currently contains a MC68881, so that no program changes will be required if the hardware is upgraded. The MC68882 offers all the features of the MC68881, as well as concurrent execution of multiple floating-point instructions, some special-purpose hardware for faster format conversions, simultaneous access to the floating-point registers by the conversion and arithmetic processing units, and reduced coprocessor interface overhead. All these contribute to increased throughput. Please see the *Motorola MC68881/MC68882 Floating Point Coprocessor User's Manual* for details of the programming differences (which are relatively minor).

Here are some rules about writing the MC68881 directive:

- If you include one or more operands, the Assembler will change only these characteristics specified by those operands from their previous values.
  - If you do not include any operands, the Assembler will reset all four characteristics to their default values: COID=1, ROUNDING=N, PRECISION=X, and KFACTOR=-16.
  - Operands may be written in any order.
- ◆ *Note:* The Macintosh ROM contains routines that perform a variety of fixed-point mathematical operations. For information about these routines, see the Toolbox Utilities chapter of *Inside Macintosh*.

---

### MC68851: Assemble MC68851 coprocessor instructions

[*macro-label*] MC68851

This directive tells the Assembler that subsequent source text may contain instructions to an MC68851 Paged Memory Management Unit coprocessor. Your source text must contain this directive before the first such instruction.

- ▲ **Warning** You may not use an MC68851 directive in the same assembly with a MACHINE MC68030 directive. ▲

---

### STRING: Specify string format

[*macro-label*] STRING  $\left\{ \begin{array}{l} \text{ASIS} \\ \text{PASCAL} \\ \text{C} \end{array} \right\}$

The STRING directive tells the Assembler how to encode all string constants occurring in the data-definition directives DC and DCB and in literals. The Assembler encodes strings as specified until it processes the next STRING directive. STRING PASCAL is the preset condition.

You can supply any one of these three operands with `STRING`:

Operand	Effect
<code>ASIS</code>	Strings are encoded exactly as specified; they contain just the characters included between the single quotation marks.
<code>PASCAL</code>	Pascal-formatted strings are generated. Each one is preceded by a length byte, as if it were stored in a Pascal variable of the type <code>STRING</code> .
<code>C</code>	C-formatted strings are generated. This format always contains at least one 0 byte following the last character of the string.

The Assembler may add one or more 0 bytes after the last character of any string, to end it on a word or long word boundary. Literals and strings defined by `DC .w` are filled to the next word boundary; strings defined by `DC .L` are filled to the next long word boundary.

---

### BRANCH and FORWARD: Resolve forward branches

<code>[ macro-label ]</code>	<code>BRANCH</code>	$\left\{ \begin{array}{l} \text{S[SHORT]   B[BYTE]} \\ \text{W[WORD]} \\ \text{L[LONG]} \end{array} \right\}$
------------------------------	---------------------	---

<code>[ macro-label ]</code>	<code>FORWARD</code>	$\left\{ \begin{array}{l} \text{W[WORD]} \\ \text{L[LONG]} \end{array} \right\}$
------------------------------	----------------------	--

`BRANCH` and `FORWARD` tell the Assembler what size to assume for the displacement encodings of forward-referenced identifiers. `BRANCH` covers the branch instructions `BCC`, `BSR`, and `BRA`, if no size specification (S or L) is given with the mnemonic. `FORWARD` covers the base and outer displacements for the MC68020 extended addressing modes 6 and 73, shown in Table 3-2. Size specifications are discussed in Chapter 3 under "Forward-Reference Addressing."

`BRANCH WORD` and `FORWARD WORD` are the preset conditions; they generate 16-bit displacements. If you specify S, `SHORT`, B, or `BYTE`, the Assembler generates 8-bit displacements. If you specify L or `LONG` it generates 32-bit displacements. Each `BRANCH` or `FORWARD` directive remains in effect until the next `BRANCH` or `FORWARD`. You can specify `FORWARD LONG` with MC68020 instructions only.



- ◆ *Note:* The Assembler will report an error on any forward-referencing instruction with too small a displacement field. For example, if `BRANCH S` is specified but the Assembler generates a displacement that is too great for eight bits, the Assembler will report an error.

---

### OPT: Specify level of code optimization

```
[ macro-label ] OPT { ALL
                     NONE
                     NOCLR }
```

The Macintosh Workshop Assembler accepts generic forms for certain machine instructions and addresses, converting them to other forms at run time. The instructions for which it accepts generic forms are listed in Appendix A; the address formats are listed in Table 3-2. There are three general reasons for making such conversions:

- **Optimization:** The Assembler converts instructions and addresses if they can be encoded more efficiently. The result occupies less memory and often runs faster as well. An example of an instruction conversion is `SUBA An, An` in place of `MOVE #0, An`. Examples of address conversions are `bd(PC)` for `(bd, PC)` and the suppression of MC68020-addressing base displacements and outer displacements when their values are 0.
- **Convenience:** The Assembler converts instructions on the basis of their context—for example, `ADDI` in place of `ADD`. It also permits substituting instructions to make coding easier and more readable—for example, by substituting `BZ` for `BEQ`.
- **Compatibility:** The MPW Assembler converts certain instructions to make them compatible with other assemblers. Examples include substituting `BHS` for `BCC` and `BLO` for `BCS`.

The generic address forms listed in Table 3-2 are all converted for optimization. The generic instruction forms listed in Appendix A are grouped by their reasons for conversion.

The `OPT` directive lets you control parts of this conversion process. You might want to eliminate optimization, for example, when writing a table of branch instructions or a routine with a critical execution time.

You can specify one of three operands:

- **ALL** allows all conversions. This is the preset case.
- **NONE** eliminates all optimizations; the Assembler will not accept generic instruction forms and will not optimize addressing modes.
- **NOCLR** is similar to **ALL**. On some hardware, a **CLR *ea*** instruction in place of **MOVE #0, *ea*** is not exactly equivalent. **NOCLR** provides for this difference by allowing all conversions except for the **MOVE-TO-CLR** substitution.

The current **OPT** directive setting remains in effect until the next **OPT** directive is processed.

---

### CASE: Specify treatment of lowercase letters

```
[ macro-label ] CASE { ON | Y[ES] }  
                   { OFF | N[O] }  
                   { OBJ[ECT] }
```

The **CASE** directive lets you determine how the Assembler interprets lowercase letters in identifiers.

The operand **ON**, **Y**, or **YES** forces the Assembler to treat uppercase and lowercase letters as distinct. For example, the Assembler treats **abcd** and **Abcd** as two different identifiers. This is the way **C** treats uppercase and lowercase letters.

The operand **OFF**, **N**, or **NO** lets the Assembler treat uppercase and lowercase letters identically. For example, the Assembler treats **abcd** and **Abcd** as the same identifier. It is the preset condition.

**OBJECT** or **OBJ** forces the Assembler to generate in the object file all module identifiers and all exported and imported identifiers exactly as specified in the source text, retaining uppercase and lowercase distinctions. It ignores uppercase and lowercase distinctions for references within the source text, however. Hence **CASE OBJECT** has the same effect as **CASE OFF** within an assembly.

- ◆ *Note:* The **CASE** directive has no effect on segment names. They are always case-sensitive except for macro variables, which are case-insensitive.

The MPW Linker always distinguishes between uppercase and lowercase when matching exported entry-point identifiers with their imported references. Hence you must be careful when linking an assembly-language program with programs written in C or Pascal. When `CASE OFF` is in effect, the Assembler generates exported and imported identifiers entirely in upper case. This matches the MPW Pascal Compiler, which also generates uppercase identifiers. When `CASE ON` is in effect, the Assembler preserves the capitalization used in the source text. This matches the MPW C compiler, which maintains case distinctions.

`CASE OBJECT` lets you communicate with C in uppercase and lowercase, without needing to preserve case distinctions inside your program. Case distinctions can create a problem when you are using large files of equates—for instance, the standard Macintosh equates. `CASE OBJECT` lets you preserve case distinctions in your object file while ignoring them in your source text. The `CASE` directive remains in force until the next `CASE` directive is processed. However, it is not a good idea to mix `CASE` modes within a single source file. The `CASE` value may be overridden from the Assembler's command line with the `-case` flag.

### Writing register names

The MPW Assembler predefines two sets of all the register names listed in Table 3-4: one set all uppercase, one set all lowercase. With `CASE OFF` (the preset condition), the lowercase set of names is superfluous. With `CASE ON`, you can use both sets interchangeably. However, with `CASE ON`, the Assembler does not accept register names with any case combination except all uppercase or all lowercase; register names such as `SP` and `ZA7` are illegal. You can get around this by writing specific equates to legal predefined register names.

---

### BLANKS: Control acceptance of blanks in operand field

```
[name]          BLANKS          { ON | Y[ES] }
                                     { OFF | N[O] }
```

The `BLANKS` directive controls where the Assembler will accept spaces and tabs within the operand field. It is discussed in Chapter 2 under "Machine Instruction Syntax."

An operand of `OFF`, `N`, or `NO` lets the Assembler accept spaces and tabs only in places where the operand field is incomplete: following commas separating operand subfields and between paired constructs such as parentheses, brackets and braces. Where the operand field is complete, a space or a tab signals the beginning of the comment field.

An operand of ON, Y, or YES forces the Assembler to accept spaces and tabs anywhere in the operand field, except within single symbols (such as identifiers). With BLANKS ON, you must write a semicolon at the end of the operand field to separate it from the comment field. BLANKS ON is the preset condition.

The BLANKS directive remains in force until the next BLANKS directive is processed.

---

## Location-counter controls

The two directives described in this section control the value of the current location counter, represented in your source text by the asterisk (\*) symbol. They are the following:

ALIGN Advance the location counter to the next multiple of a value  
ORG Set the value of the location counter

---

### ALIGN: Align location counter

*[macro-label]* ALIGN *[expr]*

The ALIGN directive generally has only local effect: it forces the next code or data statement to be assembled at a new location.

When the Assembler encounters an ALIGN statement, it increments the location counter to the next multiple of the value of the ALIGN parameter *expr* (typically 2 or 4); it then continues assembling instructions and data at the next valid location for the current code or data statement. Since the Assembler aligns all instructions and any data larger than a byte on even-byte boundaries, if the location counter is odd, the Assembler will assemble the next instruction on the next even-byte boundary.

## Special cases

- ALIGN** With no parameter specified, the Assembler assumes a value of 2.
- ALIGN 0** Causes the Assembler to stop its default alignment of most data to even byte boundaries until it encounters another **ALIGN** directive with a non-zero operand. Data that are normally aligned to even byte boundaries but that are assembled on odd-byte boundaries under an **ALIGN 0** directive generate a warning.
- While the Assembler is in this no-align state, the location counter can be forced to the next even-byte boundary by a **ds .size 0** directive, where *size* is larger than B(yte). For further information, see the discussion of the **DS** directive under "Data Definitions" earlier in this chapter.
- ALIGN 1** Has no effect, except to start assembler default alignment again, if it has been turned off by **ALIGN 0**.

The statement **ALIGN *a-expr*** acts the same as **ORG *o-expr***, where  $o-expr \bmod a-expr = 0$ . The parameter *expr* may not contain any forward, undefined, or imported references. Except for the special case of an *expr* value of 0, **ALIGN** directives may appear only inside code modules, data modules, or templates.

Remember these points when using **ALIGN**:

- When you use **ALIGN** in decrementing templates and data modules, the Assembler decrements the location counter. In other words, it aligns in the same direction as the prevailing counter direction.
- If you use **ALIGN** with no operand, the Assembler assumes a value for *expr* of 2, thereby aligning the current location counter to the next word boundary.

---

## ORG: Set location counter

*[macro-label]*    ORG            *[expr]*

ORG sets the current module or template location counter to the value specified by the expression *expr*.

Remember these points when using ORG:

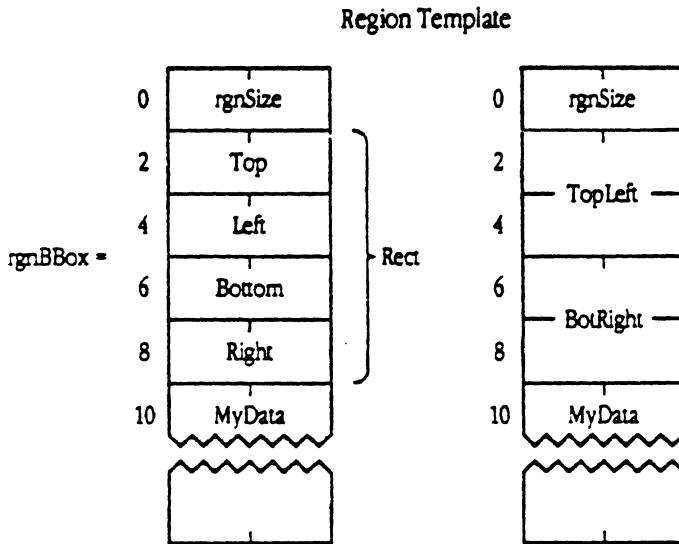
- You cannot use ORG to change the location counter from positive to negative (or vice versa) in code modules, data modules, or imported templates. You can change the sign of the location counter only in nonimported templates.
- You must be careful when using ORG to set the current location counter backward in a decrementing template. Remember that in a decrementing template each label is defined by first decrementing the location counter and then assigning its value to the label. You must take this additional decrement into account.
- If you use ORG with no operand, the Assembler sets the current location counter to the maximum positive or maximum negative location-counter value assigned to the module up to this point. The Assembler sets the current location counter to the maximum positive value for code modules and for incrementing templates and data modules. It sets the current location counter to the maximum negative value for decrementing templates and data modules.

The following example illustrates the use of ORG in a template definition:

```
Region            RECORD        0
rgnSize          DS.W            1            ; Integer
rgnBBox          EQU            *            ; Rect
Top              DS.W            1
Left             DS.W            1
Bottom          DS.W            1
Right            DS.W            1
                 ORG            Top            ; Points mapped over Rect
topLeft         DS.L            1
botRight        DS.L            1
                 ORG                            ; Make sure of location counter
MyData          DS.B            100          ; Reserve 100 bytes
                 ENDR
```

The template just defined has the memory format shown in Figure 4-2.

■ Figure 4-2 Sample template format



In this example, the first `ORG` directive sets the location counter to `Top`, so that `topLeft` and `botRight` map onto `Top`, `Left`, `Bottom`, and `Right`. The second `ORG` directive has no operand, so the Assembler sets the location counter to the highest value so far—in this case, 10. Although the mapping is exact, using `ORG` guarantees a correct final value for the location counter. This technique is particularly useful, for example, when a variant field does not exactly map onto another variant.

---

## File controls

The file control directives described in this section let you create and access files other than the current source text files during assembly. The directives are as follows:

**INCLUDE**    Insert source text from another file  
**DUMP**        Write the current global symbol table to a file  
**LOAD**        Read a file into the current global symbol table  
**ERRLOG**     Create an error-listing file

These directives are discussed in detail later in this chapter. An additional file control directive, `MACLIB`, is reserved for future implementation.

---

## File search rules

The Assembler searches in several directories for files specified in `INCLUDE` and `LOAD` directives. If the directive is given a full pathname (a name containing at least one colon but not beginning with a colon), it opens the specified file. If the directive is given a partial pathname (a name that either starts with a colon or contains no colons), it searches the accessible directories for the file in the following order:

1. The current directory
2. The directory that contains the current input file
3. The directory or directories specified by the `-i` Assembler option, in the order specified. Assembler options are described in Appendix G.
4. The directory or directories specified in the `{AIncludes}` MPW Shell variable. Shell variables are discussed in *Macintosh Programmer's Workshop Reference*.

The foregoing search rules are implemented by prefixing the specified partial filename with the name of the directory being searched.

---

## INCLUDE: Take source text from another file

*{macro-label}*      `INCLUDE`      *filename*

The `INCLUDE` directive causes the Assembler to accept source input from a specified file. The value of *filename*, a quoted literal string, is the name of the file. The file is said to be **included**. Here is an example:

```
INCLUDE 'traps.a'
```

The Assembler takes input from the included file until it reaches the end of the file. It then resumes taking input from the original file, starting with the line following the `INCLUDE` directive. The only time the Assembler does not switch back to the original file is when it encounters an `END` directive in the included file.

An included file may itself include another file. Included files may be nested in this way up to five levels deep. When looking for included files the Assembler follows the procedure described above under "File Search Rules."

`INCLUDE` directives are not permitted in macro definitions.



---

## DUMP and LOAD: Write and read symbol table files

```
[macro-label]      DUMP      filename  
[macro-label]      LOAD      filename
```

The `DUMP` and `LOAD` directives let you store and retrieve the Assembler's global symbol tables in external files. This capability helps speed assembly by letting the Assembler access often-used symbol tables from a file instead of building them repeatedly. Symbol tables stored in an external file are said to be "dumped." When they are retrieved, they are "loaded."

The value of *filename* is the name of an external file. With the `DUMP` directive, the Assembler creates a new file, or overwrites an old file, of that name. With the `LOAD` directive, the Assembler searches for the specified file according to the rules given earlier in this chapter under "File Search Rules."

Symbol tables are created by the Assembler from the source text. They are kept in memory different lengths of time, depending on the scope of the symbols in the table. Local symbol tables for code modules are purged at the end of the assembly of each module. Global symbol tables—containing module identifiers, data module field identifiers, and all symbols defined outside of code modules, including macro definitions—are kept for the duration of the assembly process.

By using `DUMP`, you can tell the Assembler to write the following items from the current global symbol table to the file named by *filename*.

- template definitions (including type information)
- absolute equates
- equates to imported identifiers
- register equates
- `OPWORD` definitions
- imported identifiers
- macro definitions

Remember that `LOAD` assumes the same environment as when `DUMP` was used. The Assembler does no cross-checking to determine whether, for instance, register equates for floating-point registers are defined even though `MC68881` is not turned on in the "loaded" file.

DUMP does not write these items:

- macro variables
- data module identifiers
- data module field definitions
- data type information
- code module identifiers
- code module labels
- information about exported identifiers

You use `LOAD` to read a previously dumped symbol table into current memory. When a dumped symbol table is loaded, the loaded symbols are merged with the current global symbol table. This means that there must not be duplicate definitions; the Assembler will report, as an error, any conflict between a current symbol table entry and a loaded entry. A symbol entry in a loaded table will not override an already existing symbol entry. You can use the `LOAD` directive only outside modules and templates.

The Assembler writes files containing dumped symbol tables in a format that is more compact than the original source files used to produce the symbol tables. Hence it is advantageous to use `DUMP` and `LOAD` when you are using a large number of equates—for example, with the standard Macintosh equates. If you do, your dumped files will require substantially less disk space than the corresponding source files. The assembly process will go faster as well, because the Assembler will not need to scan all the external equate source files.

---

### ERRLOG: Specify error log file

*[macro-label]*    `ERRLOG`    *filename*

The `ERRLOG` directive lets you create a separate log file containing all the error messages reported by the the Assembler. Its name will be the value of *filename*. All errors and warnings reported in the listing file will also be copied to this error log file. At the start of the assembly process, there is no preset error log file, unless one is specified by the `-e` option in the Assembler command string. You specify one with the first `ERRLOG` directive in your source text. However, the file is not actually created until there is an error or warning message to write into it.

- ◆ *Note:* If only warnings (no errors) are generated during assembly, the error log file is not created.

You can also switch error log files by specifying another `ERRLOG` directive with a different filename. You can terminate error logging by using `ERRLOG` with a null string for its filename.

◆ *Note:* Use of this option under MPW is discouraged.

---

## Listing controls

The listing control directives let you control the layout and content of the listing file that the Assembler produces during the assembly process. They include the following directives:

<code>PAGESIZE</code>	Specify the listing page size
<code>TITLE</code>	Define a title for the listing header
<code>PRINT</code>	Control miscellaneous listing options
<code>EJECT</code>	Start a new page in the listing
<code>SPACE</code>	Insert blank lines in the listing

These directives are discussed in detail below. In addition, you can control the listing font and font size by using the `-font` Assembler option described in Appendix G.

The assembly listing format is described in Appendix C.

---

### **PAGESIZE: Specify listing page size**

`[macro-label] PAGESIZE [lines][, width]`

The `PAGESIZE` directive lets you specify the number of lines and the number of characters per line that the Assembler sends for each page of the listing. Both the *lines* and *width* parameters must be absolute expressions; they cannot contain any forward, undefined, or imported references.

The *lines* parameter indicates how many text lines the Assembler sends to the listing file between successive form feed characters (ASCII \$0C). Its value must be greater than 29. Each page also contains six header lines; therefore the actual page length is *lines* + 6. If you omit `PAGESIZE` in your source text or include `PAGESIZE` without a *lines* value, the Assembler will assume a default value of 75 text lines. With the six header lines, this makes a default page length of 81 lines. The *width* parameter indicates how many characters the Assembler sends to the listing file between successive return characters (ASCII \$0D). Its value must lie in the range 70..160. The Assembler uses this number to right-justify the date and page entries in the header and to truncate the source line display in the listing. If you omit `PAGESIZE` in your source text or include `PAGESIZE` without a *width* value, the Assembler will assume a default line width of 126 characters. The default values for *lines* and *width* are based on printing listings in 7-point Courier on a LaserWriter® printer. On screen, the listing is presented in 7-point Monaco.

---

### **TITLE: Specify title line for listing**

`[macro-label] TITLE str-expr`

The `TITLE` directive lets you specify a title line to be placed in the header information of the listing file. This title remains in effect until the next `TITLE` directive. The text *str-expr* may contain a maximum of 80 characters; the Assembler will truncate it if it is either longer than 80 characters or too long to fit in the header.

When the Assembler encounters a `TITLE` directive in your source text, it performs an `EJECT` action to terminate the current listing page. The new title appears on the next page, with the `TITLE` directive listed first below the new page's header.

---

### **PRINT: Control listing information**

`[macro-label] PRINT parameter,...`

The `PRINT` directive lets you control whether or not the Assembler creates a listing, and if so, what information it contains. It may contain from 1 to 13 parameters, separated by commas, in the operand field. The parameters may appear in any order. The possible parameters for any `PRINT` directive are shown in Table 4-3 and explained below; those that are underlined in Table 4-3 are preset. Those values are in force unless you specifically override them.

■ Table 4-3 PRINT directive parameters

Parameter	Action
<u>ON</u>	Send lines to the assembly listing file
<u>OFF</u>	Do not send lines to the assembly listing file
<u>GEN</u>	Show macro expansions
<u>NOGEN</u>	Do not show macro expansions
<u>PAGE</u>	Allow automatic page ejects
<u>NOPAGE</u>	Suppress automatic page ejects
<u>WARN</u>	Show warning messages
<u>NOWARN</u>	Do not show warning messages
<u>MCALL</u>	Show macro call statements
<u>NOMCALL</u>	Do not show macro call statements
<u>OBJ</u>	Show generated object code
<u>NOOBJ</u>	Do not show generated object code
<u>DATA</u>	Show up to 90 bytes of generated data
<u>NODATA</u>	Show only the first line of generated object data
<u>MDIR</u>	Show macro directive lines
<u>NOMDIR</u>	Do not show macro directive lines
<u>HDR</u>	Show header lines
<u>NOHDR</u>	Do not show header lines
<u>LITS</u>	Show generated literals
<u>NOLITS</u>	Do not show generated literals
<u>STAT</u>	Show assembly status
<u>NOSTAT</u>	Do not show assembly status
<u>SYM</u>	Show symbol tables
<u>NOSYM</u>	Do not show symbol tables
<u>PUSH</u>	Save current print status
<u>POP</u>	Retrieve saved print status

ON allows a listing file only if you specified a listing filename when invoking the Assembler. OFF suppresses listing until PRINT ON occurs. PRINT OFF directives are not listed in the listing file, regardless of any other parameters they may contain.

GEN and NOGEN control the listing of macro expansion lines. Macro directives appear only if PRINT MDIR is also in force.

PAGE and NOPAGE control whether or not the Assembler sends automatic form feed characters to the listing file.

WARN and NOWARN control both the display and counting of warning messages.

`MCALL` and `NOMCALL` control the listing of macro call statements.

`OBJ` lets the Assembler list the generated object code or data for each listed line. Generated object code is always shown in full. Up to 18 lines (about 90 characters) of generated object data are shown if `PRINT DATA` is also in force. If `PRINT NODATA` is in force, only one line of generated object data is shown. `NOOBJ` suppresses all listing of object code and data. This results in a briefer listing that shows only source text lines and their addresses. If you include `PRINT NOOBJ` in your source text it should be the first line, to avoid changes in format after the listing has begun.

`DATA` and `NODATA` control whether object data is shown in full or limited to one line. These parameter values are effective only if `PRINT OBJ` is in force.

`MDIR` and `NOMDIR` control whether or not macro directives (including conditional-assembly directives and `SETA` and `SETC` directives) are shown in the listing. `PRINT GEN`, `NOMDIR` lets you list macro expansions without listing any of the macro control statements that produced them.

`HDR` and `NOHDR` control whether or not header lines are printed in the listing. `HDR` is effective only if `PRINT PAGE` is in force.

`LITS` and `NOLITS` control the listing of literals produced by `PEA` and `LEA` machine instructions. If `LITS` is in force, literals are printed at the end of the code module in which they were produced.

`STAT` and `NOSTAT` control showing the assembly status in the listing. If `PRINT STAT` is in force, each module identifier is listed to the diagnostic output file when the Assembler encounters it in the source text.

`SYM` and `NOSYM` control showing symbol tables in the listing, including local symbol tables at the end of each module and the global symbol table at the end of the assembly.

`PUSH` saves the current `PRINT` parameter values before processing new ones. `POP` retrieves the most recently saved values and places them in force. You can save up to five different sets of values in this way. `PRINT` directives containing `PUSH` or `POP` are not listed. A typical use of `PUSH` and `POP` is to save the current listing format before listing a macro in a different format, then restore it afterward. By using the `&SETTING` function described in Chapter 6, you can access the listing format at any level on the stack.

To override the preset values of one or more parameters, write a `PRINT` directive in your source text containing only the parameter values you want to change. Repeating a parameter value already in force will not cause an assembly error.

- ◆ *Note:* You can also accomplish most of the `PRINT` directive actions by using the `-print` Assembler option described in Appendix G. However, `PRINT` directives in your source text override the `-print` Assembler option.

---

### **EJECT: Start new listing page**

*[macro-label]*    `EJECT`            *[lines]*

`EJECT` causes the next line of a listing to appear at the top of the next page. The `EJECT` directive itself is not listed. If `EJECT` occurs when the next line would already be at the top of the next page, it has no effect.

The parameter *lines* is optional. If present, it must be an absolute expression without any forward, undefined, or imported references, with a positive value greater than zero. *Lines* makes `EJECT` conditional on whether the specified number of lines is available on the current page. If the lines are available, the Assembler takes no action; if not, it ejects a new page in the listing. Using this parameter lets you make sure that a section of your source text is listed all on one page.

---

### **SPACE: Insert blank line in listing**

*[macro-label]*    `SPACE`            *[lines]*

`SPACE` lets you insert one or more blank lines into your listing. The `SPACE` directive itself is not shown. The optional parameter *lines* indicates how many blank lines to insert (from one to the current page length). It must be an absolute expression without any forward, undefined, or imported references. If the parameter is omitted, the Assembler inserts one blank line. If the value of the *lines* parameter exceeds the number of lines remaining on the page, `SPACE` has the same effect as `EJECT`.





## Part II The Macro Processor and the Macro Language

EVERY LINE OF YOUR SOURCE TEXT IS INTERPRETED BY THE Macro Processor before it is handed to the rest of the Assembler. The Macro Processor operates on the elements of the **macro language**: macro directives, macro variables, which may be present within or outside of macros, and conditional assembly directives. The next three chapters describe this **macro language** and its use in detail. ■

- Chapter 5 tells you how to write **macro definitions** and **macro calls**. A **macro definition** is a named section of source text containing the statements or directives that constitute a macro. Macro definitions are set off by the directives `MACRO` and `ENDM` or `MEND`. A **macro call** is a statement that invokes a macro by name, causing the Macro Processor to **expand** the call. When the Macro Processor expands a macro call, it replaces the macro call statement with the contents of the macro definition, substituting actual values for certain of its variables and parameters. Macro expansion makes it easy for you to generate lengthy but repetitious source text sequences, by defining a few macros and then calling them repeatedly.
- Chapter 6 discusses **macro variables** and the functions that operate on them. A macro variable is a variable whose value is assigned by the macro language. The macro language contains a set of functions that let you form expressions out of constants and macro variables. When such expressions occur in the body of a macro, they are replaced by their current values every time the Macro Processor expands a call to that macro. You can use macro

variables outside of macros (if they were originally declared to be global) and thereby access their current values outside or inside macros. There are three kinds of macro variables: **symbolic parameters** (discussed in Chapter 5), **SET variables**, and **Assembler system variables**. These three types differ in the ways they acquire values. Symbolic parameters acquire values when the Macro Processor expands macro calls; Assembler system variables are assigned values by the Assembler itself; and SET variables are assigned values by explicit macro directives.

- Chapter 7 describes the MPW Assembler macro directives, which are instructions you give to the Macro Processor. Using the macro directives, you can determine whether the Assembler will process or ignore sections of your source text, based on the values of boolean control expressions. This facility is called **conditional assembly**. It is a powerful tool for creating and controlling source text structures.

## Chapter 5 **Macros**

A **MACRO** IS A PREVIOUSLY DEFINED SEQUENCE OF STATEMENTS, directives, or both that the Assembler processes when it encounters a corresponding **macro call**. During macro processing, the Assembler usually generates new source text. This chapter describes the part of the macro language that is involved in defining and using macros. ■

### *Contents*

Macro expansion	117
Scope of macro symbols	118
Defining macros	118
MACRO and ENDM or MEND: Delimit macro	119
The prototype statement	119
The macro body	120
Macro comments	121
Symbolic parameters	123
Concatenating symbolic parameters	124
Calling macros	125
The macro-qualifier	126
Macro call labels	127
Operand syntax	128
Paired single quotation marks	128
Paired parentheses and brackets	128
Ampersands	129
Commas	129
Blanks (spaces and tabs)	129
Backquotes	130
@-labels	130
Omitted or extra operands	130

Operand sublists	131
Accessing sublist elements	131
Parameter types and default values	132
Nesting macros	133
Keyword macros	135
Defining keyword macros	135
Calling keyword macros	136
Mixed-mode macros	138

---

## Macro expansion

Each line of source text is handled by the Macro Processor as follows:

- If the source text line does not contain any elements of the macro language, the Macro Processor simply passes it unaltered to the rest of the Assembler for assembly-language interpretation.
- If the line contains any macro variables or functions but is not a macro directive statement, the Macro Processor replaces the expressions with their current values. It then passes the result to the rest of the Assembler for assembly-language interpretation.
- If the line is a macro directive statement, the Macro Processor handles it according to the rules defined in Chapter 7. In this case it interprets and acts upon any macro expressions that the line contains, instead of replacing them with their current values.

Thus macro directive statements are handled entirely by the Macro Processor; they are not passed to the rest of the Assembler. Machine instruction statements and other directive statements are handled by the rest of the Assembler, after the Macro Processor has converted any macro expressions they might contain to actual values.

You should remember certain rules that the Macro Processor follows when preparing statements to be passed to the rest of the Assembler:

- An element to be replaced, such as a macro parameter or a variable reference, may not be continued across a line boundary.
- An element to be replaced may not contain another element (such as an array index or function parameter) that must be replaced first.
- All fields are expanded, including comments. If you want to refer to a macro variable in a comment that is part of a macro call or prototype statement, you must precede it by a **backquote** (```) to force the Macro Processor to treat it literally instead of replacing it with its value. Similarly, the name of a macro function must be preceded by an extra ampersand.

---

## Scope of macro symbols

As explained in Chapter 2 in "Scope of Definitions," the scope of a definition is the range of source text in which the defined identifier can be accessed by code or data statements. A variable is called "accessible" within the scope of its identifier. The lifetime of a variable is the duration of the assembly process over which it is accessible. These rules govern the scope of macro symbols and the lifetime of macro variables:

- Variables declared within a macro, including all its parameters, are accessible only in subsequent statements of that macro. Their lifetime is the duration of the current expansion of the macro; they are not accessible in future or nested invocations of the same macro. Local variables with the same identifier that are defined in other macros or outside of macros are different variables.
- Variables declared local outside of any macro definition are accessible only in subsequent statements outside macros. You can think of them as program-level variables. Their lifetime is the remainder of the assembly process after their definition. Local variables of the same identifier defined inside macros are different variables.
- Variables declared global within a macro are accessible in all subsequent statements of the program. Their lifetime is the remainder of the assembly process after their definition.
- When the lifetime of a local variable or the scope of its identifier overlaps that of a global variable, the local variable takes precedence and the global variable becomes temporarily inaccessible.

---

## Defining macros

A macro definition is a sequence of statements that tells the Macro Processor the name of a macro, the format of its call, and the source text to be generated when the macro is called. It consists of four parts:

- the **header** directive (`MACRO`)
- the **prototype** statement
- the macro **body**
- the **trailer** directive (`ENDM` or `MEND`)

Here is the format and syntax of a macro definition:

```
MACRO
Header
[label]    name,macro-qualifier    [parameter-list]
Prototype
          machine instruction or directive statements
          Body
[macro-label]    { ENDM | MEND }
Trailer
```

Here are some rules about macro definitions:

- Every macro must be defined before it can be used; that is, the definition must precede any calls to the macro.
- You can write macro definitions anywhere in a program except within other macro definitions.
- A macro whose definition appears within a conditional section of source text (for example, within a section delimited by the `IF` directive) will not be defined if the conditional branch causes the Macro Processor to skip over its definition.

---

## MACRO and ENDM or MEND: Delimit macro

The `MACRO` and `ENDM` directives begin and end the macro definition. In place of `ENDM` you may write `MEND`.

`MACRO` takes no labels or operands. `ENDM` may be preceded by a macro label, but such a label can be referred to only by a `GO TO` directive (described in Chapter 7).

---

## The prototype statement

The macro prototype statement specifies the name of the macro being defined and the format of calls to the macro. The prototype statement also establishes the identifiers of the macro parameters, if any. The basic form of the macro prototype statement is as follows:

Label field	Operation field	Operand field
[label]	name,macro-qualifier	[parameter-list]

For the full syntax of the macro prototype statement, including parameter types, default values, and keywords, see "Parameter Types and Default Values" in "Calling Macros," later in this chapter.

The only required part of the macro prototype statement is the macro's name, which identifies it for later calls. The macro name must follow the rules for identifiers given in Chapter 2. It may not be the same as the name of any machine instruction, Assembler directive, or other macro used in the same assembly.

In the other parts of the prototype statement—the label, the macro qualifier, and the **parameter list**—you may write only symbolic parameter identifiers. These are valid assembly-language identifiers preceded by an ampersand (&). They are described later in this chapter, in "Symbolic Parameters." For the parameter list, you may write a series of symbolic parameter identifiers separated by commas. The macro qualifier, if present, must follow the macro name with a period as a separator.

The label field in a macro prototype statement is discussed in "Macro Call Labels," later in this chapter.

You may continue a prototype statement on the next source text line at any point after the macro name (or macro qualifier, if present). To continue a line, you must break it after a comma and insert a backslash as a continuation character, as in the following:

```
MyMacro    &parmA, &parmB, &parmC, \  
           &parmD, &parmE
```

You can continue a prototype statement indefinitely.

---

## The macro body

The macro body consists of the set of machine instruction or directive statements between the macro's prototype statement and its trailer. It may contain any or all of three kinds of source lines:

- **model statements**, which the Macro Processor uses as a model to generate actual Assembler statements or directives
- **macro directives**, which control the process of macro expansion but generate no statements
- **inner macro calls**, which invoke other macros

◆ *Note:* You can nest macros recursively to a maximum depth of 512.



Just like other assembly-language statements, model statements consist of four fields, some of which may be omitted: the label field, the operation field, the operand field, and the comment field.

The label field may be omitted or may contain a symbol or symbolic parameter.

- ◆ *Note:* A symbolic parameter in a macro label field whose *value* is an asterisk (\*), period-asterisk (.\*), or semicolon (;) will not turn a model statement into a comment, since the Macro Processor decides whether the statement is a comment before substituting values for its macro variables.

The operation field may contain any MPW assembly-language instruction or directive, including macro directives, or any variable symbol. It may not, however, contain an INCLUDE directive. If it contains the word *endm* or *mend*, the macro will terminate at that point.

The operand field within a model statement follows the same rules as in other statements and directives. The operand fields of some macro directives, however, require embedded keywords. In such cases the operand field terminates when the required syntax is complete, rather than with the first space or tab.

---

## Macro comments

Several types of comments can be added to macros. Comments that appear on macro prototype statements, macro model statements, and lines of their own are expanded with the macro. Special macro comment statements, which are present only in the macro definition, are also permitted.

You may add comments to macro prototype statements, subject to these rules:

- If your prototype statement has no parameters, you must begin your comment with a semicolon.
- If a parameter list is present and you want to write a comment, you must separate your comment from the end of the parameter list by at least one space or tab (with **BLANKS ON**) or a semicolon (with **BLANKS OFF**).

- If the parameter list is continued on a second line, you may insert comments on individual lines. Finish each line with a comma and a backslash continuation character. The Macro Processor will treat all text remaining on the line after the continuation character as a comment. Here is an example:

```

MyMacro  &parmA,          \ Comment on ParmA
         &parmB, &ParmC, \ Comments on ParmB and ParmC
         &parmD          \ Comment on ParmD

```

You may add comments to macro model statements in the same way as you add them to ordinary statements, but you must follow one additional rule. To refer to a macro variable or function in a comment, you must precede it with an additional ampersand; otherwise the Macro Processor will substitute its value during macro expansion.

You can write a line containing only a comment in a macro if it starts with an asterisk (\*) or a semicolon (;). Additionally, you may write a special variety of comment line beginning with period-asterisk (.\*), which is not stored in the macro definition. In an Assembler listing, a comment beginning with a period-asterisk will be listed in the macro definition but not in any macro expansion; comment lines that begin with a sole asterisk or a semicolon will be listed both in the definition and in all expansions.

The following example demonstrates all the types of macro comment lines. The line numbers are for reference only.

```

1      MACRO
2          DbgHead                ; Macro DebugHead
3      .* Puts the Pascal entry code in front of a
4      .* subroutine so MacsBug can identify it
5      * >>> Debug Header <<<
6          LINK      A6,#0        ; Set up stack frame
7      ENDM

```

The comment on line 2 is preceded by a semicolon because the prototype statement has no parameters. The comments on lines 3 and 4 explain what the macro does. They are intended to appear only in the listing of the definition, so they are preceded by period-asterisks. The comment on line 5 is a note that appears in the expansion of each call, so it is preceded by an asterisk alone. This causes the Macro Processor to list it both in the definition and in all expansions. The comment on line 6 is a model statement comment.

---

## Symbolic parameters

Symbolic parameters are a special type of variable symbol, to which the Macro Processor assigns values when a macro is called. They are used to pass information to the macro from macro call statements. Symbolic parameters must be defined in the prototype statement. Only then can you refer to them in statements in the macro body. The values of symbolic parameters are assigned when a macro is called and cannot be subsequently changed except by another call to the same macro.

The following macro definition illustrates the use of symbolic parameters. The line numbers are for reference only; they would not appear in an actual source text.

```
1      MACRO
2      Incr      &src,&dest ; Get src, increment, move to dest
3      MOVE.W   &src,D0   ; Move &src to temp register
4      ADDQ.W   #1,D0     ; Increment temp register
5      MOVE.W   D0,&dest  ; Move temp register to &&dest
6      ENDM
```

When the macro `Incr` is called, the Macro Processor replaces all references to symbolic parameters in the macro body by corresponding values in the macro call (except when the reference is preceded by an additional ampersand). Thus when the Macro Processor encounters the macro call

```
      Incr      Alpha,Beta
```

it replaces it with the following lines:

```
3      MOVE.W   Alpha,D0   ; Move Alpha to temp reg
4      ADDQ.W   #1,D0     ; Increment temp reg
5      MOVE.W   D0,Beta   ; Move temp reg to &&dest
```

Notice that the `&src` parameter defined on line 2 of the macro definition appears twice on line 3 in the macro body. When the macro is expanded, both references to `&src` are replaced with the parameter value `Alpha`. In line 5, however, the second occurrence of `&dest` is not replaced with the value `Beta`, because it is preceded by a second ampersand. The process of macro expansion is explained in more detail later in this chapter in "Calling Macros."

## Concatenating symbolic parameters

When symbolic parameters occur outside macro directives, and are not part of SET variable subscripts or function arguments, you can concatenate them with other characters or symbols by simply putting the objects to be concatenated next to each other, without any intervening spaces. Potential ambiguities arise when a character, a number, a left square bracket, a period, or an ampersand is concatenated to the right of a symbolic parameter reference, because the Macro Processor interprets these characters as part of the parameter reference. In such cases you must terminate the parameter reference with a period, to distinguish the end of the parameter identifier from the characters concatenated to the right.

When a symbolic parameter or a variable reference is followed by a period, the Macro Processor replaces the symbol and the period with the symbol's value when the macro is expanded. The period does not appear in the generated statement.

The following table shows some sample results from the concatenation of a parameter with various combinations of text, other parameters, and special characters. The column on the left contains examples of concatenation with the parameter `&param`. The column on the right shows the result of macro expansion when the value of `&param` is A:

Expression	Result
<code>&amp;param.B</code>	AB
<code>&amp;param..B</code>	A.B
<code>&amp;param.(B)</code>	A(B)
<code>B&amp;param</code>	BA
<code>B,&amp;param</code>	B,A
<code>B2&amp;param</code>	B2A
<code>&amp;param.2B</code>	A2B
<code>&amp;param,.2B</code>	A,.2B
<code>&amp;param&amp;param</code>	AA
<code>&amp;param.&amp;param</code>	AA
<code>&amp;param..&amp;param</code>	A.A

The following examples illustrate some variations that often cause problems:

Expression	Result	Notes
<code>&amp;param.[1]</code>	A[1]	Period makes it not a sublist reference
<code>&amp;param[1]</code>	null	Sublist reference, but parameter value is not a list, yields null string
<code>&amp;param.B</code>	AB	
<code>&amp;paramB</code>	error	If <code>&amp;paramB</code> is not defined

The first two examples look like attempts to write parameter sublist references, as described in "Calling Macros"; they also have the same form as subscripted SET variable references, discussed in Chapter 6 in "Set Array Variables." The final example represents the common mistake of forgetting to terminate a parameter identifier with a period when it is followed by other characters, causing the Assembler to read the string as one parameter name, &paramB.

---

## Calling macros

A macro call is an instruction to the Macro Processor to insert at that point in the source text, the statements or directives specified by the macro definition. The inserted source text replaces the macro call statement.

The format and syntax of a macro call is as follows:

Label field	Operation field	Operand field	Comment field
[ <i>label</i> ]	<i>macro-name</i> [ <i>macro-qualifier</i> ]	[ <i>parameter-list</i> ]	[ <i>comment</i> ]

The contents of the operation, operand, and comment fields of a macro call are summarized in the next few paragraphs; the content of the label field is discussed in the next section.

The operation field contains the name of the macro to be called. The macro being called must have been defined previously in the assembly process.

The operand field contains information that the macro call passes to the body of the macro. Thus the order of operands in a macro call statement must correspond to the order of the symbolic parameters in the prototype statement of the corresponding macro definition. Parameters of this type are called **positional parameters**. The parameters defined in a macro definition are called **formal parameters**, while the corresponding parameters specified in a macro call are called **actual parameters**. The Macro Processor assigns the values of the actual parameters to the formal parameters when it calls a macro.

When **BLANKS OFF** is in effect, operands specified in a macro call must be separated by commas, with no intervening spaces. The first space not embedded inside a quoted string will terminate the operand field (as well as the parameter list) and begin the comment field. With **BLANKS ON** in effect, leading and trailing blanks in the operand field are ignored and comments must be preceded by semicolons. The form required for individual operands in both cases is discussed in detail later in this chapter.

The comment field is ignored by the Macro Processor. The operand and comment fields of macro calls may be continued on more than one line, using the conventions applicable to prototype statements. For details, see "The Prototype Statement" earlier in this chapter.

---

## The macro-qualifier

The *macro-qualifier* is a way to allow a macro to act as much as possible like a single instruction. That is, just as certain instructions may be qualified with a size value (byte, long, word, and so on), the *macro-qualifier* allows a macro to accept similar size qualifications. It is the responsibility of the macro to check the information passed through for correctness.

The following example illustrates a macro call with a *macro-qualifier* defined:

```
MACRO
BMOVE.&size      &src,&dest,&len
---
ENDM

BMOVE           source,destination,12
BMOVE.B        source,destination,12
BMOVE.L        source,destination,3
END
```

When the BMOVE macro is invoked, the value of `&size` is undefined (blank) in the first case. In the second case it is "B", and in the third case, it is "L". (The period after the macro name is stripped out by the preprocessor.)

Unlike other parameters, the *macro-qualifier* does not take default values. In the example given here, the value of `&size` has to be tested explicitly against blank.

---

## Macro call labels

The label field of the macro call may contain an identifier. The Macro Processor interprets macro call labels according to these rules:

- If the prototype statement of the macro being called does not contain a label, the macro call label refers to the current location counter value at the point of the macro call. In code modules, this is a reference to the first generated statement of the macro. In data modules, however, the current location counter is not necessarily aligned with the next defined data item.
- If the prototype statement of the macro being called contains a label, the value of the call's label is passed to the macro, as if the prototype's label were an operand. However, the syntax of the label identifier is restricted to the rules covering other identifiers; you cannot declare it as a sublist or keyword, type it, or give it a default value, as you can with a normal operand.
- When the value of the call's label is passed to the macro as just described, it may be used for any purpose—as a label on a statement in the body of the macro, or as a variable.

For example, suppose a call to the macro `INCR`, described earlier in this chapter in "Symbolic Parameters," is contained in the following code fragment (the line numbers are included for reference):

```
          BRA.S      x1
          - - -
x1      INCR      D2,D3
```

Because the macro definition of `INCR` does not contain a label in its prototype statement, the branch to `x1` will automatically be interpreted as a branch to the first generated statement of `INCR` (line 3).

Now suppose the same call is used with a different definition of `INCR`:

```
1      MACRO
2      &y1  INCR      &src,&dest  ; Get src, increment, move to dest
3      MOVE.W   &src,D0      ; Move &src to D0
4      ADDQ.W   #1,D0        ; Increment D0
5      &y1  MOVE.W   D0,&dest   ; Move D0 to &&dest
6      ENDM
```

In this case the branch will go to line 5, because the actual value `x1` is now passed to the formal label `&y1`, which is a label on the third generated line of `INCR`.

---

## Operand syntax

The value of any macro call operand may be numeric or may be any sequence of up to 255 characters. In the latter case, you must observe these conventions:

### Paired single quotation marks

A macro call operand may contain one or more strings enclosed by single quotation marks ('), called **quoted strings**. However, quoted strings may themselves contain single quotation marks. Single quotation marks that are part of the string are written as two adjacent single quotation marks. Thus, if the first single quotation mark of a quoted string is numbered one, then the string ends with the first even-numbered single quotation mark that is not followed immediately by another. The first and last single quotation marks of a quoted string are called **paired single quotation marks**.

The following sample operand consists of a sequence of characters that contains two quoted strings:

```
'='10'C''C's
```

The first and second single quotation marks are paired, as are the third and last. The Macro Processor interprets the fourth and fifth single quotation marks as one embedded single quotation mark. Note that this sample operand contains characters in addition to those in the strings.

### Paired parentheses and brackets

A macro call operand must be balanced with respect to parentheses and brackets; that is, there must be an equal number of left and right parentheses and the  $n$ th left parenthesis must appear to the left of the  $n$ th right parenthesis. The same is true for square brackets.

The simplest case of **paired parentheses** is a left parenthesis followed by a right parenthesis without any intervening parentheses or brackets. Similarly, the simplest case of **paired brackets** is a left bracket followed by a right bracket without any intervening parentheses or brackets. If there is more than a single pair of parentheses or brackets, the Macro Processor associates them by repeatedly recognizing and removing such simple pairs.

This method is used because paired parentheses and brackets normally enclose lists, which may be nested. Searching for the simplest pairs lets the Macro Processor recognize the list structure, as well as determine whether any given comma is an interior part of a list or the end of an operand in the parameter list.



The following example contains three sets of paired parentheses and one set of paired brackets:

```
(READ, (src.text, EXT)) inBufx (input {2, 512})
```

The first and fourth parentheses are paired, as are the second and third and the fifth and sixth. The two brackets are paired.

The Macro Processor ignores any parentheses or brackets appearing between paired single quotation marks when associating paired parentheses, as in the following example:

```
(N' ('closed)
```

### Ampersands

To write a literal ampersand (&) in source text, you must write two consecutive ampersands. The Macro Processor interprets any single ampersand as the beginning of a reference to a symbolic parameter, a SET variable or function call, or a Macro Processor system variable. In nested macro calls in which a symbolic parameter is concatenated to one or more preceding ampersands, the Macro Processor interprets the concatenation as a sequence of literal ampersands in which the final odd-numbered ampersand is the beginning of a variable reference.

### Commas

A comma delimits the end of every macro call operand unless the comma appears between paired single quotation marks, parentheses, or brackets. For example, the following is a single operand even though it contains several commas:

```
(D0, D1, D2) delim', 'right
```

### Blanks (spaces and tabs)

The effect of spaces and tabs in a macro call operand depends on the setting of the `BLANKS` Assembler directive. If the current setting is `BLANKS OFF`, then any space or tab terminates the operand field, unless the line is continued. The only exceptions are spaces or tabs written between paired single quotation marks, as in the following example:

```
'Value too large' (&Count, 10)
```

If the current setting is `BLANKS ON`, the Macro Processor will retain spaces and tabs if they are followed by more valid operand characters. It will ignore trailing spaces and tabs if they occur at the end of a line or before the comma delimiting the next operand.

## Backquotes

The backquote (``) is used in macro prototype and call operands to indicate that the next character should be passed through the Macro Processor without interpretation. Thus an ampersand may be put into a call operand without the Macro Processor interpreting it as the beginning of an & reference if it is preceded by a backquote. For example:

```
Bonanno` &Sons
```

A backquote can be used to literalize any character in a macro prototype or call operand, including another backquote.

## @-labels

When interpreting macro call operands, the Macro Processor assumes that any string beginning with @ is an @-label, unless it is enclosed in single quotation marks or preceded by a backquote. For example, when processing the sublist (A, @2, B) the Macro Processor will treat the element @2 as an @-label. Implementation restrictions force it to make this assumption in order to encode the scope of the caller's @-label inside the macro. If you want to begin a call operand that is not an @-label with @, you must literalize it with a backquote.

## Omitted or extra operands

If you omit an actual parameter from a macro call, you must still include the comma that would have separated the omitted actual parameter from the next parameter. This preserves the positional correspondence of parameters. The Macro Processor gives each omitted actual parameter a value of 0 if the corresponding formal parameter has integer type, or assigns it the null string if the corresponding formal parameter is type string. Parameter types are discussed in Chapter 6.

If you omit one or more of the last operands from a macro call, you may also omit the commas that would have separated these final operands. With macros that use the number of actual parameters internally, the Assembler counts trailing commas when determining the number.

These rules are demonstrated in the following sample macro prototype statement and its subsequent macro call:

```
Prototype  ExampleMac  
           &parm1, &parm2, &parm3, &parm4, &parm5, &parm6  
Call      ExampleMac D0, *+6, , 'syntax error'
```

In this macro call, the third parameter has been omitted (nothing is between the second and third commas), as have the fifth and sixth parameters; no final commas are required.

To avoid having the Macro Processor assign the null string as the value of an actual parameter omitted from a macro call, you may write the prototype statement so that it assigns default values to omitted parameters. "Parameter Types and Default Values," later in this chapter, tells you how to do this.

It is permissible in a macro call to specify more actual parameters than there are formal parameters defined in the corresponding macro definition. There will, of course, be no formal parameter identifiers by which you can refer to these extra operand values; however, you can still access them by using `&SYSLIST`, as described in Chapter 6 in "Assembler System Variables."

---

## Operand sublists

You can structure a macro call operand as a **sublist**. This lets you refer to a collection of operands in the same way as you would refer to a single operand, while still being able to refer to individual members of the collection. Each operand in a sublist is called an **element**. Sublists are always string types.

A sublist may contain one or more operands, separated by commas and enclosed in paired parentheses. The Macro Processor treats the entire sublist, along with the paired parentheses, as a single operand (limited to 255 characters). Any element of a sublist may itself be a sublist. Some examples are shown here:

```
(A)
(A, B, C)
(A, (B, C), D)
((A), B, C)
```

A sublist may be continued on subsequent lines and may contain comments, following the same conventions as macro prototype statements. However, it may not contain more than 255 characters (excluding comments). These conventions are described in "The Prototype Statement," earlier in this chapter.

## Accessing sublist elements

You can refer to an element of a macro sublist operand in the body of the macro. To do this, you write a sublist as an absolute arithmetic expression enclosed in square brackets immediately after the identifier of the sublist's symbolic parameter. The expression in brackets represents the **index** (position) of the desired element, with the first element being 1. For example, if `&abc` is the identifier of a sublist parameter, then `&abc[n]` refers to the *n*th element in the sublist. The index value *n* must be greater than zero.

If the  $n$ th element of the sublist is omitted, or if there are fewer than  $n$  elements in the sublist, then the value of `&abc [n]` will be the null string. If the  $n$ th element of the sublist is itself a sublist, then `&abc [n,m]` refers to the  $m$ th element in the sublist which is the  $n$ th element of `&abc`. You can create sublist references up to any depth by specifying as many subscripts as necessary, separated by commas, between the brackets. You can use the `&NBR` function, described in Chapter 6, to find out how many elements a sublist has.

If you write a sublist reference to a parameter that is not a sublist, its value will always be the null string.

Note that the left bracket that begins the index expression must follow the parameter identifier without any intervening characters. You should not place a period between the parameter identifier and the left bracket unless you want to concatenate the value of the entire operand with a left bracket.

The following examples illustrate various references to sublist elements. They are based on a parameter `&p` with the sublist `(A, (B, C), D)` as its value. The left column shows sublist element references; the right column shows their corresponding values.

Reference	Value
<code>&amp;p</code>	<code>(A, (B, C), D)</code>
<code>&amp;p[1]</code>	<code>A</code>
<code>&amp;p[2]</code>	<code>(B, C)</code>
<code>&amp;p[3]</code>	<code>D</code>
<code>&amp;p[4]</code>	<code>null</code>
<code>&amp;p[2,1]</code>	<code>B</code>
<code>&amp;p[2,2]</code>	<code>C</code>

### Parameter types and default values

The Macro Processor assumes that all macro parameters are strings. Sometimes, however, it is necessary to declare a parameter as an integer type. You can assign a parameter a type by following the parameter identifier in the prototype statement with a colon (`:`) followed by `STR` or `C` for a character string parameter and `INT` or `A` for an integer (arithmetic) parameter.

If you omit an actual parameter of type integer when calling a macro, the Macro Processor gives it a value of 0. If you omit a string parameter, the Macro Processor assigns it the null string.

The actual value passed to a parameter of integer type must be an integer constant, because the Macro Processor interprets it using the `&STRTOINT` function. Values of integer expressions must be passed as strings and then converted inside the macro, using the `&EVAL` function. The functions `&STRTOINT` and `&EVAL` are described in Chapter 6.

You can assign default values to macro parameters by following their identifiers (or type specifications) in the prototype statement with an equal sign (=) followed by the default value. Each such parameter will take on the default value if it is omitted in a macro call.

The following example of a macro prototype statement defines two parameters of different types with different default values; `&sz` is an integer with a default value of 4, while `&reg` is a string with a default value of 'D0'.

```
MACRO
BumpSz      &sz: Int=4, &reg=D0
- - -
ENDM
```

The full syntax of any macro prototype statement is therefore the following:

$$[ \& \textit{name} \quad \textit{name} . \& \textit{name} \quad \left[ \& \textit{name} : \left\{ \begin{array}{l} \text{INT} \\ \text{STR} \\ \text{A} \\ \text{C} \end{array} \right\} \quad \left[ \{ \text{--} \} \textit{opnd-value} \right] , \dots \right]$$

The double equal sign (==) preceding the default operand value *opnd-value* in this syntax diagram identifies keyword parameters. Keyword parameters are discussed in "Keyword Macros" in this chapter.

---

## Nesting macros

Macros may be called from the bodies of other macro definitions. Such macro calls are termed **inner macro calls**, because they are executed within a macro definition. They are also termed "nested calls." A macro call that is not within the body of a macro definition is an **outer macro call**.

When the Macro Processor encounters an inner macro call during macro expansion, it suspends processing the current macro and expands the inner macro. When it has finished expanding the inner macro, it resumes expansion of the outer macro. The macros invoked by inner macro calls may call other macros, so there may be any number of macros up to 511 (the maximum nesting depth is 512) suspended in midprocess while the current macro is being expanded. The sequence of macros suspended in this way is called the current **macro call chain**.

The definition of a macro called by an outer macro call may contain any number of inner macro calls. The outer call is termed the **first-level call**; all inner calls in a first-level macro are termed **second-level calls**; calls within second-level macros are **third-level calls**; and so on. The number of each such **dynamic nesting level** is available in the Assembler listing if macro expansions are being listed.

Nesting levels refer only to the way the Macro Processor actually expands macros in a given assembly. Any particular inner macro call may have different nesting levels at different points in an assembly, because the macro containing it may be called from a variety of places, along a variety of call chains. The number of levels of nested macros that your program can call depends only upon the complexity of your macro constructs and the memory available.

The following example demonstrates how a macro may be called from within another macro. This is the definition of `Incr`, the macro to be called:

```
MACRO
Incr      &src,&dest
MOVE.W   &src,D0
ADDQ.W   #1,D0
MOVE.W   D0,&dest
ENDM
```

Now here is the definition of a macro, `Incr2`, that calls `Incr`:

```
MACRO
Incr2    &a,&b,&c
.* Increment &a and &b, put sum in &c
Incr     &a,&a
MOVE.W   &a,D1
Incr     &b,&b
ADD.W    &b,D1
MOVE.W   D1,&c
ENDM
```

If `Incr2` is called with `x`, `y`, and `z` as its three parameter values, the Macro Processor will generate the statements shown in the following code segment by macro expansion. The line numbers on the left are included only for reference to the example.

```

Incr2    X,Y,Z      ; macro call
1        MOVE.W     X,D0      ; macro expansion
2        ADDQ.W     #1,D0
3        MOVE.W     D0,X
4        MOVE.W     X,D1
5        MOVE.W     Y,D0
6        ADDQ.W     #1,D0
7        MOVE.W     D0,Y
8        ADD.W      Y,D1
9        MOVE.W     D1,Z
```

Notice that lines 1 through 3 are generated by the first inner macro call to `INCR`. Lines 5 through 7 are generated by the second inner macro call. The other lines are generated from the `INCR2` macro.

---

## Keyword macros

Keyword macros provide an alternate way to pass parameter values during macro expansion. In a keyword macro definition, you can specify parameters in any order and give them default values. The Macro Processor identifies different parameters by their keywords rather than by their position in the parameter list. As a result, when you call a keyword macro, you need specify only those parameters that require values different from their default values.

The different ways you can write macros are distinguished by this terminology:

- **Positional macros** are the kind discussed earlier in this chapter. Their parameters are distinguished by their positions in the macro prototype statement.
- **Keyword macros** are the kind described in the section that follows this list.
- **Mixed-mode macros** contain both positional and keyword parameters. They are described at the end of this chapter.

◆ *Note:* You cannot use `&SYSLIST` to access keyword parameters.

---

## Defining keyword macros

The prototype statement that defines a keyword macro is like the prototype statement for a positional macro (described at the beginning of this chapter in “Defining Macros”), except that each parameter identifier in the parameter list is followed by two equal signs (`==`). This tells the Macro Processor that the parameter identifiers are **keywords**.

The two equal signs may optionally be followed by an integer constant or string giving the parameter’s default value. This constant must have the same type as the parameter.

The following are examples of valid keyword prototype operands:

```
&recSize==12
&recSize:INT==12
&inLine==
&err==(12, 'syntax error')
```

The following are examples of invalid keyword prototype operands:

InLine	No & (not a parameter)
&recSize	No equal signs
&blkSize ==512	Space before ==
&x=2	Single equal sign; interpreted as a positional operand with a default value

The following is an example of a keyword prototype statement with a symbolic parameter in the label field and three keyword parameters in the operand field:

```
&lab CopyBuff &src==InBuff,&dest==,&count==512
```

Note that the second parameter does not have a default value.

---

## Calling keyword macros

Once you have defined a keyword macro, you can tell the Macro Processor to expand it and insert it into your source text with a **keyword macro call** directive of the following form:

```
[label]          macro-name    [keyword=[value] ] ,...
```

The keyword operands are keywords without their & prefixes, each of which is followed immediately by an equal sign (=). Each equal sign may optionally be followed by a value. Such operand values must conform to the same rules as operand values in positional macro calls (see "Calling Macros," earlier in this chapter).

The keywords specified in the macro call must have the same identifiers as the keyword parameters defined in the keyword macro definition (without initial ampersands). The Macro Processor does not distinguish between uppercase and lowercase in keywords.

The following are examples of valid keyword macro call operands:

```
RecSize=1024
dest=printBuf
Count=
```

The following are examples of invalid keyword macro call operands:

&Err=(8,'bad input')	Starts with &
sysIn =foo.text	Space before =
dest	No equal sign
=1024	No keyword



Here are some rules about writing keyword macro calls:

- You can write keyword operands in any order.
- You need specify only those operands whose values must be different from the default values specified in the macro definition.
- You need not write extra commas for omitted operands.

When the Macro Processor expands a keyword macro, it follows these rules:

- It processes identifiers in the label and operation fields in the same way that it processes such identifiers in positional macro calls.
- In the operand field, it replaces all parameters mentioned in the call with their specified values.
- It replaces all parameters not mentioned in the call with their default values, as specified by the keyword macro prototype statement.
- If a parameter not mentioned in the call has no default value, the Macro Processor replaces it with the null string (type string) or 0 (type integer).
- If a parameter is mentioned in the call but is not followed by a value, the Macro Processor replaces it with the null string (type string) or 0 (type integer).

The following example illustrates the use of keyword macros. It is the same as the example given in "Symbolic Parameters" in this chapter but rewritten with keywords instead of positional parameters. Using keywords improves the macro in four ways: it lets you specify a default temporary register, it gives you the option of leaving the result in the temporary register by simply not specifying a destination register when you call the macro, makes it easy for you to specify an increment value other than 1, and makes the macro call easier to understand.

```
MACRO
&lbl   Incr      &src==, &dest==, &DReg==D0, &inc==1
&lbl   MOVE.W   &src, &DReg
        ADD.W    #&inc, &DReg
        IF      '&dest' <> '' THEN
        MOVE.W   &DReg, &dest
        ENDF
        ENDM
```

The following are examples of possible calls to this macro:

```
Incr      src=D2, dest=D1
Incr      src=D3, inc=4
Incr      src=D0, dreg=D1
```

---

## Mixed-mode macros

**Mixed-mode macros** are macros that contain a combination of positional and keyword parameters.

The prototype statement of a mixed-mode macro resembles that of a positional macro except for its parameter list. In a mixed-mode macro parameter list, you must write all the keyword parameters after all the positional parameters. A positional parameter may not follow a keyword parameter.

The operand lists of mixed-mode macro call directives may include zero or more positional operands, plus zero or more keyword operands. All the actual positional parameters must precede the first actual keyword parameter.

Remember these points when using mixed-mode macros:

- You treat positional operands as if they were in a positional macro and keyword operands as if they were in a keyword macro.
- If you omit a positional parameter, you must insert a comma to indicate the missing position; however, you may omit all trailing commas.
- You can nest all three kinds of macros—positional, keyword, and mixed-mode—in macros of the other kinds. In other words, a macro of any kind may be called as an inner macro from a macro of the same or any other kind.
- The Assembler system variable `&SYSLIST` lists only the positional parameter values in a mixed-mode macro call.

The example given earlier in "Calling Keyword Macros" could be changed to a mixed-mode macro by removing the equal signs after the `&src` and `&dest` parameters in the prototype statement. Then the keyword macro call,

```
Incr      src=A3,inc=4
```

could be replaced with this mixed-mode macro call:

```
Incr      A3,inc=4
```

In this example, writing a mixed-mode macro definition lets you specify the frequently used `&src` and `&dest` parameters more conveniently in your macro calls, without having to write out their keywords.

## Chapter 6 Macro Variables and Functions

MACRO VARIABLES ARE VARIABLES OF THE MACRO LANGUAGE. You can combine them with the functions described in this chapter and use them to control conditional assembly plus certain features of macro expansion. They include the following:

- symbolic parameters, which acquire values when the Macro Processor expands macro calls
- SET variables, which are assigned values by explicit macro directives
- assembler system variables, which are assigned values by the Assembler itself

Symbolic parameters were discussed in Chapter 5. SET variables and Assembler system variables are discussed in this chapter. ■

### *Contents*

SET variables	141
SET variables and symbolic parameters	143
LCLA, LCLC, GBLA, and GBLC: Define SET variables	143
SETA and integer expressions	145
&ABS: Return absolute value	146
&EVAL: Evaluate contents of string	147
&ISINT: Test string for integer content	147
&LEN: Measure string length	147
&LEX: Parse string lexically	148
&LIST: Divide string into list	150
&MAX: Find maximum in integer list	151
&MIN: Find minimum in integer list	151
&NBR: Count sublist elements	151
&ORD: Return integer value	152
&POS: Find position of substring in string	152
&SCANEQ and &SCANNE: Scan string	153
&STRTOINT or &S2I: Convert string to integer	154

- Symbol table functions 154
  - &NEWSYMTBL: Create new symbol table 154
  - &ENTERSYM: Enter or update symbol in table 155
  - &FINDSYM: Find symbol in table 156
  - &DELSYMTBL: Delete symbol table 157
- SETC and string expressions 157
  - Accessing substrings of string variables 158
  - &CHR: Convert integer to character 159
  - &CONCAT: Concatenate strings 160
  - &DEFAULT: Return string value or default 160
  - &GETENV: Return MPW Shell variable value 160
  - &INTTOSTR or &I2S: Convert integer to string 160
  - &LOWCASE or &LC: Convert string to lowercase 161
  - &SETTING: Return directive setting 161
  - &SUBSTR: Return substring of string 162
  - &TRIM: Trim spaces and tabs from string 163
  - &TYPE: Determine identifier type 163
  - &UPCASE or &UC: Convert string to uppercase 164
- SET array variables 165
  - Defining SET array variables 165
  - Using SET array variables 166
  - Accessing substrings in SET array string elements 167
- Assembler system variables 168
  - &SYSINDEX or &SYSNDX: Macro call index 168
  - &SYSLIST or &SYSLST: Macro operand list 169
  - &SYSSEG: Current segment identifier 170
  - &SYSMOD: Current module identifier 170
  - &SYSDATE: Current date 170
  - &SYSTIME: Current time 170
  - &SYSTOKEN and &SYSTOKSTR: Values set by &LEX 171
  - &SYSVALUE and &SYSFLAGS: Values set by &FINDSYM 171
  - &SYSLOCAL and &SYSGLOBAL: System symbol table ID's 171

---

## SET variables

SET variables help you program the ways that the Macro Processor converts your macro structures into the source text that the rest of the Assembler assembles.

SET variables can have either integer or string values. The MPW Assembler Macro Processor recognizes two different groups of directives, one to handle integer SET variables and the other to handle string SET variables. Similarly, it interprets two classes of expressions formed from these variables. It treats the Boolean expressions used to control conditional assembly directives as a restricted class of integer expressions.

You use separate directives to define SET variables and to assign them values. The four **variable definition directives** are shown here:

LCLA	Defines integer SET variables of local scope (LoCaL Arithmetic)
GBLA	Defines integer SET variables of global scope (GloBaL Arithmetic)
LCLC	Defines string SET variables of local scope (LoCaL Character)
GBLC	Defines string SET variables of global scope (GloBaL Character)

The two directives that assign values to SET variables are shown here:

SETA	Assigns a value to an integer set variable
SETC	Assigns a value to a string set variable

The MPW Assembler macro language lets you form complex expressions out of macro variables, using a variety of **built-in functions**. The functions that return integer values are shown here:

&ABS	Returns an absolute value
&ISINT	Tests whether a string contains an integer
&LEN	Returns the length of a string
&LEX	Scans the tokens in a string
&LIST	Divides a string into a list and places the list elements in an array
&MAX	Returns the largest value in a list of integer expressions
&MIN	Returns the smallest value in a list of integer expressions
&NBR	Returns the number of elements in a sublist or parameter list
&ORD	Returns the integer value of a relocatable numeric expression or one-character string expression
&POS	Finds the position of a substring in a string
&SCANEQ	Finds a character in a string
&SCANNE	Finds the first non occurrence of a character in a string
&STRTOINT	Converts a string expression to its integer value

Among the integer functions, the following let you create and manipulate symbol tables:

<b>&amp;NEWSYMTBL</b>	Creates a new symbol table and returns its id number
<b>&amp;ENTERSYM</b>	Enters or updates a symbol in a symbol table
<b>&amp;FINDSYM</b>	Finds a symbol in a symbol table
<b>&amp;DELSYMTBL</b>	Deletes a symbol table

The functions that return string values are shown here:

<b>&amp;CHR</b>	Converts an integer value to a one-character string containing its ASCII equivalent
<b>&amp;CONCAT</b>	Concatenates string expressions
<b>&amp;DEFAULT</b>	Returns a default string value if a test string is null
<b>&amp;GETENV</b>	Returns the value of an MPW Shell variable
<b>&amp;INTTOSTR</b>	Converts an integer expression to its string value
<b>&amp;LOWCASE</b>	Converts a string to all lowercase
<b>&amp;SETTING</b>	Returns the current setting value for certain other directives
<b>&amp;SUBSTR</b>	Returns a substring of a string
<b>&amp;TRIM</b>	Removes leading or trailing spaces and tabs from a string
<b>&amp;TYPE</b>	Returns the type of a symbol
<b>&amp;UPCASE</b>	Converts a string to all uppercase

One function returns either an integer or string value but is ordinarily used as an integer function:

<b>&amp;EVAL</b>	Converts a string expression to either an integer or a string, by evaluating the expression contained in its string argument
------------------	--

The Macro Processor treats SET variables differently, depending on where they occur in the source text. It follows these rules:

- When a SET variable appears in the label, operation, or operand field of a model statement in a macro definition, the Macro Processor replaces the SET variable with its current value regardless of its context. In this situation, SET variables act like symbolic parameters. Remember that SET variables must be declared in the macro definition (redeclared if they are declared outside the macro definition) in order for you to use them in the macro.
- When a SET variable appears inside a macro definition as a label or operand of a macro directive statement, it is replaced only if explicitly allowed by the syntax of the directive or expression where it occurs. In this situation, SET variables act like normal variables.
- SET variables may also appear outside macro definitions, but only in SET variable definitions, SETA or SETC directives, absolute expressions, immediate effective addresses, and conditional assembly directives. If they appear in other kinds of expressions, the Assembler does not recognize them as SET variables.

---

## SET variables and symbolic parameters

SET variables have broader capabilities than the symbolic parameters discussed in "Defining Macros" in Chapter 5. These are their main differences:

- Flexibility of values. Symbolic parameters are assigned values in macro call statements, and these values remain fixed during macro expansion. SET variables acquire values when you use either the `SETA` or `SETC` directive. Symbolic parameters are not referred to as variables because their values cannot be changed during macro expansion. SET variable values, on the other hand, may vary during macro expansion as the result of any number of `SETA` or `SETC` directives.
- Use outside macros. Symbolic parameters can be accessed only inside macro definitions. SET variables can be accessed both inside and outside of macros. SET variables outside macros are normally used to control conditional assembly.
- Different types. Unless you type them to the contrary in their prototype statement, symbolic parameters are always strings. SET variables, on the other hand, are divided explicitly into string and integer types. Each type has its own functions for creating expressions.
- Global or local scope. Symbolic parameters always have local scope; they may be accessed only in the macro in which they are defined. You can define SET variables so that their scope is local to a macro. However, you can also define SET variables so that their scope is local to your source text outside any macro. Finally, you can define them globally, so that the same SET variable may be accessed in several macros or both inside and outside macros.

---

### LCLA, LCLC, GBLA, and GBLC: Define SET variables

```
[ macro-label] { LCLA  
                { LCLC  
                { GBLA  
                { GBLC  
                set-var-name, ...
```

You must define any SET variable before you can give it a value or refer to it in an expression. You use the `LCLA`, `LCLC`, `GBLA`, and `GBLC` directives to define SET variables. `GBLA` and `GBLC` define global integer and string variables, respectively, and `LCLA` and `LCLC` define local integer and string variables.

Each of these directives takes a list of one or more SET variable identifiers (*set-var-name* in the syntax diagram just given), separated by commas, in its operand field. Every SET variable identifier consists of an ampersand (&) followed by a valid identifier (as if the ampersand were a valid initial alphabetic character), just like a symbolic parameter identifier. If the SET variable is an array its identifier is followed by a dimension, as described later in this chapter in "Defining SET Array Variables." The Macro Processor ignores uppercase and lowercase distinctions in these identifiers, regardless of the setting of the CASE Assembler directive. LCLA, LCLC, GBLA, and GBLC directives may appear anywhere in your source text, including in macro definitions.

These are examples of valid SET variable definitions:

```
LCLA      &n
LCLC      &args
GBLC      &mainModName, &mainSegName
```

You decide whether to use integer-type or string-type SET variables by analyzing the values the variable must store and how the variable is to be used. Integer variables contain 32-bit values in the range -2147483648..+2147483647. String variables contain strings of length 0..255. Such strings may contain any ASCII characters, with these exceptions:

- String constants may not contain the return character (ASCII code \$0D).
- String constants may not contain the NUL or SOH characters (ASCII codes \$00 and \$01) when appearing in macro definitions, because these characters have special significance during macro definition storage.

The choice of whether to use variables of local or global scope is determined by where a variable needs to be accessed. Local scope is sufficient for most purposes, unless a variable must be shared by several macros or preserved between macro calls. The applicable rules are given in Chapter 5 in "Scope of Macro Symbols."

Remember these rules when defining SET variables:

- You cannot use the same identifier for both a symbolic parameter and a SET variable in the same macro. For example, consider the following macro prototype statement:

```
&label    MyMove      &from, &to
```

The Assembler would report an error if you tried to define a SET variable named `&label`, `&to`, or `&from` within the macro—that is, in the same scope.

- You cannot define two SET variables with the same identifier but with different types (integer and string) and with the same or overlapping scopes.
- A global variable defined by a GBLA or GBLC directive must be redeclared whenever it is used in a particular macro.



The Macro Processor assigns **initial values** to SET variables when they are defined, following these rules:

- Integer types are set to 0.
- String types are assigned the null string (length zero).
- Local variables are given initial values whenever they are defined—that is, whenever a LCLA or LCLC directive occurs.
- Global variables are given initial values only the first time they are defined—that is, when the first GBLA or GBLC statement defining the variable occurs.

---

## SETA and integer expressions

Every SET variable of integer type has a 32-bit value. The Macro Processor sets this value to 0 at the first definition of a global integer variable and at every definition of a local integer variable. You can subsequently change the value, using the SETA directive. It has this form:

*set-var-name*                      SETA                      *arith-expr*

The label *set-var-name* must be the identifier of an integer SET variable defined by LCLA or GBLA that includes the SETA directive in its current scope. The operand *arith-expr* must be an absolute integer expression. It may contain any of the following functions, described in this section:

&ABS	Returns an absolute value
&DELSYMTBL	Deletes a symbol table
&ENTERSYM	Enters or updates a symbol in a symbol table
&EVAL	Converts a string expression to either an integer or a string, depending on its actual value
&FINDSYM	Finds a symbol in a symbol table
&ISINT	Tests whether a string expresses an integer
&LEN	Returns the length of a string
&LEX	Scans the tokens in a string
&LIST	Returns number of elements in a list created from a string
&MAX	Returns the largest value in a list of integer expressions
&MIN	Returns the smallest value in a list of integer expressions

<b>&amp;NEWSYMTBL</b>	Creates a new symbol table and returns its id number
<b>&amp;NBR</b>	Returns the number of elements in a sublist or parameter list
<b>&amp;ORD</b>	Returns the integer value of an integer expression or one-character string expression
<b>&amp;POS</b>	Finds the position of a substring in a string
<b>&amp;SCANEQ</b>	Finds a character in a string
<b>&amp;SCANNE</b>	Finds the first nonoccurrence of a character in a string
<b>&amp;STRTOINT</b>	Converts a string expression to its integer value

Integer expressions used with **SETA** must conform to the syntax rules for integer expressions set forth in Chapter 2 in "Expressions." They are also subject to these additional rules:

- As absolute expressions, they must not contain any forward, undefined, or imported references.
- They may contain references to integer **SET** variables and to symbolic parameters and nonmacro variables and constants with integer values, as long as their scopes include the **SETA** directive.
- They may refer to untyped symbolic parameters with string values, as long as the string values express valid integer constants. Symbolic parameters of explicit string type must be converted to integer type, using the **&STRTOINT** function described later in this chapter.

These are examples of valid integer expressions:

```
33
lab + 12
tableBase + (2 * &entryNum)
```

---

### **&ABS: Return absolute value**

**&ABS** (*arith-expr*)

The **&ABS** function returns the absolute value of the integer expression *arith-expr*. Hence if *arith-expr* is negative, **&ABS** returns its positive value. The Macro Processor does not check for value overflow.

---

## **&EVAL: Evaluate contents of string**

**&EVAL** (*str-expr*)

The **&EVAL** function takes a string expression argument and evaluates the contents of the string as if it were untyped. It returns either a string value or an integer value. It follows the same parsing rules as the Assembler itself; for details, see "Expressions" in Chapter 2. Here are some examples:

Function	Value returned
<b>&amp;EVAL</b> ('2 + 3')	integer 5
<b>&amp;EVAL</b> ('&foo')	'&bar' if <b>&amp;foo</b> has the value '&bar'
<b>&amp;EVAL</b> (&foo)	'fubar' if <b>&amp;foo</b> has the value '&bar' and <b>&amp;bar</b> has the value 'fubar'
<b>&amp;EVAL</b> ('2 + x')	integer 5 if x is equated to 3 (by <b>EQU</b> or <b>SET</b> )

The **&EVAL** function can be used to evaluate a macro directive parameter whose value is an expression of unknown type. For example, if the formal parameter **&a** is passed the actual value **B + 2**, the expression **&EVAL (&a)** returns the value 4 if **B = 2**, rather than the string 'B+2'.

---

## **&ISINT: Test string for integer content**

**&ISINT** (*str-expr*)

The **&ISINT** function examines the string expression *str-expr* and returns an integer value of 1 if the string expresses an integer, 0 if it does not. The rules by which **&ISINT** recognizes an integer in a string are these:

- It ignores leading and trailing spaces and tabs.
- For nonblank characters, it accepts only numerals and leading plus or minus signs.
- The integer value must not exceed 32 bits.

---

## **&LEN: Measure string length**

**&LEN** (*str-expr*)

The **&LEN** function examines the string expression *str-expr* and returns an integer value equal to the number of characters in its string value.

---

## &LEX: Parse string lexically

`&LEX (str-expr, start)`

The `&LEX` function parses a string into **tokens**—characters or substrings that the Assembler treats as syntactical units—using the Assembler's own lexical scanner. The value of `str-expr` is the string being parsed and `start` is an integer expression whose value is the scan's starting position (the first character of `str-expr` having position one). `&LEX` returns the integer value of the next token's position, or the current token position if it is the last token of the string.

Each call to `&LEX` sets the value of two Assembler system variables, as follows:

- It sets the value of `&SYSTOKEN` to an integer value that identifies the kind of token just read.
- It sets the value of `&SYSTOKSTR` to the actual characters in the token.

The possible values of `&SYSTOKEN` and `&SYSTOKSTR` are shown in Table 6-1. These variables are further discussed later in this chapter, in "Assembler System Variables."

■ **Table 6-1** Values returned by `&LEX`

<code>&amp;SYSTOKEN</code>	<code>&amp;SYSTOKSTR</code>	Meaning
0	<i>identifier</i>	Assembler identifier
1	<i>integer</i>	Assembler integer constant
2	<i>"floating-point constant"</i>	Characters enclosed in double quotation marks
3	<i>'string'</i>	Assembler string constant enclosed in single quotation marks
4	+	Plus (Assembler add symbol)
5	-	Minus (Assembler subtract symbol)
6	*	Asterisk (Assembler multiply symbol)
7	/ +	Slash or divide symbol (Assembler divide symbol)
8	//	Assembler MOD symbol
9	++	Assembler OR symbol
10	--	Assembler XOR symbol
11	**	Assembler AND symbol
12	=	Assembler equal symbol

■ Table 6-1 (continued) Values returned by &LEX

&SYSTOKEN	&SYSTOKSTR	Meaning
13	<> ≠	Assembler not equal symbol
14	<	Assembler less than symbol
15	>	Assembler greater than symbol
16	>= ≥	Assembler greater than or equal to symbol
17	<= ≤	Assembler less than or equal to symbol
18	>>	Assembler shift right symbol
19	<<	Assembler shift left symbol
20	┘	Assembler NOT symbol
21	~	Assembler one's complement symbol
22	(	Left parenthesis
23	)	Right parenthesis
24	[	Left bracket
25	]	Right bracket
27	}	Right brace
28	,	Comma
29	.	Period
30	;	Semicolon and end-of-string symbol
31	:	Colon
32	*	Number sign
33	\	Backslash (Assembler continuation character)
34		Not used
35		All other characters

By calling &LEX repeatedly, using the value returned by the previous call for each call's *start*, you can parse a string into its tokens. An &SYSTOKEN value of 30 indicates the last token in the string; &LEX sets &SYSTOKEN to 30 under these conditions:

- if &LEX finds a semicolon
- if the value of *start* is less than 1 or greater than the number of tokens in *str-exp*
- if the current setting of the BLANKS directive is OFF and &LEX finds a space or a tab

If the current setting of BLANKS is ON, &LEX ignores spaces and tabs.

- ▲ **Warning** The action of &LEX depends on the action of the Assembler's lexical scanner. This could change in future versions of the Assembler. ▲

---

## &LIST: Divide string into list

`&LIST (str-expr, str-arr[, delimiter])`

The `&LIST` function divides a string containing a list of items into an array of strings containing the individual items and returns the number of elements in the list. The value of `str-expr` is the string to be treated as a list. The value of `str-arr` is a string containing the name of a SETC array variable represented as a string expression. The `&LIST` function sets its array elements with the individual elements contained in `str-expr`. If the array is too small to hold all the elements in `str-expr`, `&LIST` returns the value of the maximum dimension of the array with a negative sign, to indicate that not all elements could be placed in the array.

The `&LIST` function treats `str-expr` as a sequence of substrings, all separated by a unique delimiter not enclosed in paired parentheses, brackets, or single quotation marks. By default, `&LIST` assumes the delimiter is a comma, but you can override this default by specifying an optional third argument to `&LIST`. This argument, `delimiter`, is a string expression that evaluates to a single character which `&LIST` is to use as the list delimiter.

Here are some examples of values returned by `&LIST`, assuming that `&a` is a SETC array variable with a maximum dimension of 4:

Function	Value returned	&a[1]	&a[2]	&a[3]	&a[4]
<code>&amp;LIST('a,b,c','&amp;a')</code>	3	a	b	c	
<code>&amp;LIST('a/b/c/d','&amp;a','/')</code>	4	a	b	c	d
<code>&amp;LIST('a,,,d','&amp;a')</code>	4	a			d
<code>&amp;LIST('(a,b)','&amp;a')</code>	1	(a,b)			
<code>&amp;LIST('a(b),([','])','&amp;a')</code>	2	a(b)	([,'])		
<code>&amp;LIST('a,b,c,d,e,f','&amp;a')</code>	-4	a	b	c	d
<code>&amp;LIST('a = b','&amp;a','=')</code>	2	a	b		
<code>&amp;LIST('(a):b','&amp;a',')')</code>	2	(a)	:b		

The last two examples show how `&LIST` can be used to split a string into two parts separated by a delimiter. In the seventh example, the delimiter is the equal sign. In the last example, a right parenthesis is the delimiter. This example illustrates how `&LIST` looks for

delimiters only when they are not enclosed between paired parentheses, brackets, or single quotation marks. The second right parentheses is not enclosed between any of these characters and is therefore valid as a possible list delimiter.

---

**&MAX: Find maximum in integer list**

`&MAX (arith-expr, ...)`

The `&MAX` function accepts a series of integer expressions, separated by commas, and returns the highest value in the list.

---

**&MIN: Find minimum in integer list**

`&MIN (arith-expr, ...)`

The `&MIN` function accepts a series of integer expressions, separated by commas, and returns the lowest value in the list.

---

**&NBR: Count sublist elements**

`&NBR ({symb-param | &SYSLIST})`

You can use the `&NBR` function to find out how many elements a sublist contains. The integer value returned by `&NBR` is equal to 1 plus the number of commas that separate operands in the sublist *symb-param*. When using the value returned by `&NBR`, keep these rules in mind:

- The empty sublist—with no elements—returns a value of 0.
- If some of the sublist elements are themselves sublists, they may contain commas. `&NBR` will not count these sub-order commas.
- If the argument given to `&NBR` is not a sublist, `&NBR` will return 0.
- If the argument given to `&NBR` corresponds to an omitted operand, `&NBR` will return 0.

The predefined variable `&SYSLIST`, which contains the actual positional parameter list of the current macro, may also be used as an argument to `&NBR`. With `&SYSLIST`, `&NBR` returns the number of positional parameters specified in the call statement that called the current macro. It does not count keyword parameters.

You can use `&NBR` with arguments that include sublist index expressions—in other words, with sublists that are elements of sublists. In those cases, the `&NBR` function returns the number of elements making up the specified sublist. For example, `&NBR(&parm[3])` returns the number of elements in the sublist `&parm[3]`. This also works with `&NBR(&SYSLIST[i])`.

---

### **&ORD: Return integer value**

`&ORD (expr)`

The `&ORD` function returns the integer value of the expression `expr`. If `expr` has a string value, it may not be more than one character long; `&ORD` returns the ASCII code of this character. If `expr` is a relocatable numeric expression, `&ORD` returns its absolute integer value. For example, `&ORD(*)`, written in a code module, lets you use the value of the location counter as an absolute value.

---

### **&POS: Find position of substring in string**

`&POS (str-expr1, str-expr2)`

The `&POS` function returns an integer value equal to the position of the first occurrence of a substring within a string. The value of `str-expr1` is the substring; the value of `str-expr2` is the string. If the substring does not appear in the string, `&POS` returns a value of 0. Otherwise, it returns the position of the first character of the substring, counting the first character of the string as 1. Here are some examples:

<b>Expression</b>	<b>Result</b>
<code>&amp;POS('bar', 'foobar')</code>	4
<code>&amp;POS('fu', 'foobar')</code>	0
<code>&amp;POS('o', 'foobar')</code>	2



---

## &SCANEQ and &SCANNE: Scan string

&SCANEQ (*ch*, *str-expr*, *start*)

&SCANNE (*ch*, *str-expr*, *start*)

The &SCANEQ and &SCANNE functions scan the string value of *str-expr* for the value of *ch*, a single-character string expression. &SCANEQ scans until it finds the same character ("scan until equal"); &SCANNE scans until it finds the first character that is not the same as *ch* ("scan until not equal").

Both functions begin scanning at the character position specified by the integer expression *start*. If *start* is positive, the string is scanned from left to right; if *start* is negative, from right to left. The first character of the string is position 1.

Both functions return an integer value equal to the number of characters skipped before finding the target character (or the first character that is not the target character); that is, they return the value of  $index - \&ABS(start)$ , where *index* is the position of the character sought. They return special values under these conditions:

- They return a value of 0 if the target character is at the *start* position.
- They return the number of characters in the value of *str-expr* if *start* is 0 or if it is positive and greater than the number of characters in the value of *str-expr*.
- They return a value of 1 if *start* is negative and its absolute value is greater than the number of characters in the value of *str-expr*.

Here are some examples of values returned by &SCANEQ and &SCANNE:

Expression	Result
&SCANNE('.', '.....foobar', 1)	5
&SCANEQ('.', '.....foobar', -11)	-6
&SCANEQ('o', '.....foobar', 1)	6
&SCANNE('o', '.....foobar', 7)	2
&SCANNE('.', '.....foobar', 100)	11
&SCANNE('.', '.....foobar', 0)	11
&SCANNE('.', '.....foobar', -100)	1

---

## &STRTOINT or &S2I: Convert string to integer

{ &STRTOINT | &S2I } (*str-expr*)

The &STRTOINT function accepts a string expression argument and converts it to an integer. This function does not accept strings beginning with "\$" or "%", the signifiers for hexadecimal and binary numbers, respectively. If the string cannot be parsed into an integer token, the Assembler will issue a warning and &STRTOINT will return a value of 0. &STRTOINT follows the same token-parsing rules as the Assembler itself; for a discussion of integer tokens, see "Numeric Constants" in Chapter 2 and the discussion of &LEX earlier in this chapter. You can also write &STRTOINT as &S2I.

---

## Symbol table functions

A special group of SET functions let you create and manipulate your own symbol tables. They are the following:

&NEWSYMTBL   Creates a new symbol table and returns its ID number  
&ENTERSYM     Enters or updates a symbol in a symbol table  
&FINDSYM      Finds a symbol in a symbol table  
&DELSYMTBL   Deletes a symbol table

These functions are described in the next four sections.

---

### &NEWSYMTBL: Create new symbol table

&NEWSYMTBL

The &NEWSYMTBL function has no parameters. It creates a new symbol table in memory and returns an integer value that is its ID number. You use this number as the value of *symtbl* in subsequent &ENTERSYM, &FINDSYM, and &DELSYMTBL expressions. You may create up to four new symbol tables at any one time, which exist in addition to the Assembler's own local and global tables (identified by the system variables &SYSLOCAL and &SYSGLOBAL). Remember, however, that each symbol table occupies a minimum of 1016 bytes of memory. &NEWSYMTBL returns a value of 0 if it cannot create a new symbol table.

---

## &ENTERSYM: Enter or update symbol in table

**&ENTERSYM** (*sym-tbl*, *symbol*, *value*, *flags*)

The **&ENTERSYM** function enters or updates a symbol in the symbol table identified by the integer expression *sym-tbl*. The value of *sym-tbl* may be **&SYSLOCAL**, **&SYSGLOBAL**, or any nonzero value returned by **&NEWSYMTBL**. If it is **&SYSLOCAL**, you can use **&ENTERSYM** only in source text inside a code or data module. The value of the string expression *symbol* is the symbol's identifier, with uppercase and lowercase distinctions preserved. If it has already been entered in the table, the existing information is overwritten.

You can use **&ENTERSYM** to create new symbol table entries or amend existing ones, either in the Assembler's own local or global tables or in a table you have created with **&NEWSYMTBL**. If you operate on one of the Assembler's tables you must follow its rules, as described later in this chapter. If you operate on a table you have created, you have more freedom. In particular, you may assign either a string or integer *value* to *symbol*, and may select any 16-bit positive *flags* value.

If the symbol table was created by **&NEWSYMTBL**, *value* may be either a string expression or a 32-bit integer expression. If it is one of the Assembler's own tables (identified by **&SYSLOCAL** or **&SYSGLOBAL**), *value* must be a 32-bit integer. In the latter case, it is used to contain the value assigned to the identifier by an equate directive (see "Symbol Definitions" in Chapter 4).

The integer expression *flags* must have a positive 16-bit value. If *sym-tbl* identifies a table created by **&NEWSYMTBL**, *flags* may have any value in the range 0..32767. You can create or update new **EQU** or **SET** definitions in the Assembler's tables with **&ENTERSYM**, using the appropriate *flags* value. If *sym-tbl* identifies one of the Assembler's tables, the *flags* value must be 0 if *value* is to be treated as assigned by an **EQU** directive or 1 if it is to be treated as assigned by a **SET** directive; no other *flags* values are allowed.

**&ENTERSYM** returns a value of 1 if it has successfully entered or updated the requested symbol table entry; 0 otherwise.

- ◆ *Note:* Before using **&ENTERSYM** to enter a new symbol, it is usually wise to check for an existing symbol of the same name by using **&TYPE**.

---

## &FINDSYM: Find symbol in table

`&FINDSYM(sym-tbl, symbol)`

The `&FINDSYM` function searches for a symbol in the symbol table identified by the integer expression `sym-tbl`. The value of `sym-tbl` may be `&SYSLOCAL`, `&SYSGLOBAL`, or any value returned by `&NEWSYMTBL`. If it is `&SYSLOCAL`, you can use `&FINDSYM` only in source text inside a code or data module. The value of the string expression `symbol` is the symbol's identifier, with uppercase and lowercase distinctions preserved.

If `&FINDSYM` cannot find the symbol, it returns a value of 0. Otherwise it returns a value of 1 and sets the values of the Assembler system variables `&SYSVALUE` and `&SYSFLAGS` equal to the `symbol`'s associated values of *value* and *flags*, respectively.

- ◆ *Note:* With regard to macros, the macro language permits you to manipulate your own or the Assembler's internal symbol tables. If you enter a symbol into your own symbol table, it is entered exactly as you present it, regardless of case sensitivity. If the Assembler enters a symbol into its own table, however, as, for example through an `EQU` directive, and case is set to off, the symbol is converted to all uppercase. The `&FINDSYM` function will then fail when it is applied to that symbol unless the symbol is presented in all uppercase. The action of `&FINDSYM` in this example is different from the action with your own symbol table. Here is an example of each:

CASE OFF

label EQU 1

```
&FINDSYM(&SYSGLOBAL, 'label')      ; fails
&FINDSYM(&SYSGLOBAL, 'LABEL')      ; succeeds

&ENTERSYM(&myowntable, 'label')
&FINDSYM(&myowntable, 'label')      ; succeeds
```

If `sym-tbl` identifies a table created by `&NEWSYMTBL`, these may be any values within the limits described earlier in "`&ENTERSYM`," including both strings and integers for the `symbol`'s *value*. If the value of `sym-tbl` is either `&SYSLOCAL` or `&SYSGLOBAL`, the following rules apply:

- The value of `&SYSFLAGS` is 0 for symbols most recently equated by `EQU` directives and 1 for symbols most recently equated by `SET` directives, except for register equates.
- The value of `&SYSFLAGS` is 2 for nonequated symbols and register equates.
- `&SYSVALUE` contains the symbol's equated value if the value of `&SYSFLAGS` is 0 or 1; otherwise it contains 0.

---

## &DELSYMTBL: Delete symbol table

**&DELSYMTBL**(*sym-tbl*)

The **&DELSYMTBL** function deletes the symbol table identified by the integer expression *sym-tbl*. The value of *sym-tbl* may be any value returned by **&NEWSYMTBL**; it may not be **&SYSLOCAL** or **&SYSGLOBAL**. The Assembler releases all memory allocated for the table, and you should not try to access it again.

**&DELSYMTBL** returns a value of 1 if the table was successfully deleted and 0 otherwise.

---

## SETC and string expressions

Every string SET variable has a variable-length string value up to 255 characters long. The Macro Processor sets this value to the null string at the first definition of a global string variable and at every definition of a local string variable. You can subsequently change the value using the **SETC** directive. **SETC** statements have this form:

*set-var-name*                      **SETC**                      *str-expr*

The label *set-var-name* must be the identifier of an accessible string SET variable—one that includes the **SETC** directive in its current scope.

The operand *str-expr* may be composed of the following:

- quoted strings
- symbolic parameters of type string
- string SET variables
- string functions

These rules apply to the operand *str-expr*:

- String functions may be combined with other string and integer functions to make more complex string expressions.
- The value of a string expression, including any intermediate values, cannot exceed 255 characters.

The operand *str-expr* may contain any of the following functions, described later in this section:

&CHR	Converts an integer value to a one-character string containing its ASCII equivalent
&CONCAT	Concatenates string expressions
&DEFAULT	Returns a default string value if a test string is null
&GETENV	Returns the value of an MPW Shell variable
&INTTOSTR	Converts an integer expression to its string value
&LOWCASE	Converts a string to all lowercase
&SETTING	Returns the current setting value for certain other directives
&SUBSTR	Returns a substring of a string
&TRIM	Removes leading or trailing spaces and tabs from a string
&TYPE	Returns the type of a symbol
&UPCASE	Converts a string to all uppercase

Here are some examples of valid string expressions:

'abacab'	String constant
&str	Symbolic parameter or SET variable
&CONCAT (&modname, '.', &var)	Function

---

## Accessing substrings of string variables

When referring to a string variable, including a string SET variable, a string parameter, or a string array element, you may append a **subscript** to access individual characters or substrings in its value. Array elements are described later in this chapter in "Set Array Variables."

There are two possible forms for a subscript that accesses part of the string value of a SET variable:

[ <i>arith-expr</i> ]	Accesses a single character in the string
[ <i>start: length</i> ]	Accesses a substring in the string

*Arith-expr*, *start*, and *length* may be absolute integer expressions of any complexity. *Arith-expr* accesses a single character, counting the first character of the string as 1. *Start* specifies the position of the first character of a substring, and *length* specifies its length. Here are some examples of accessing parts of the string value of a SET array element; they all assume that the value of the SET variable `&foo` is the string 'slangword':

Reference	Value
<code>&amp;foo[4]</code>	'n'
<code>&amp;foo[5:3]</code>	'gwo'
<code>&amp;foo[&amp;x:4]</code>	'angw' if <code>&amp;x = 3</code>

The use of a subscript with a string SET variable identifier is subject to these rules:

- When accessing a single character, if the value of *arith-expr* is greater than the length of the string, the Macro Processor will return a string consisting of a single space (ASCII \$20) character.
- When accessing a substring, if the value of *start* is 0 or is greater than the length of the string, the Macro Processor will return the null string.
- When accessing a substring, if the position of *start* is within the string, but fewer than the number of characters specified by *length* remain, the Macro Processor will return only the remainder of the string.
- You can use the substring form, but not the single character form, when accessing a macro parameter. This is because the Macro Processor interprets a single subscript as identifying an element of a sublist parameter. To access a single character in a macro parameter, use the substring form with a *length* value of 1.

---

### **&CHR: Convert integer to character**

`&CHR (arith-expr)`

The `&CHR` function returns a string value containing the single character that has the ASCII value specified by *arith-expr*. This value must lie in the range 0..255.

---

## **&CONCAT: Concatenate strings**

**&CONCAT** (*str-expr*,...)

The **&CONCAT** function concatenates the values of any number of string expressions, separated by commas. The value returned by **&CONCAT** is the string resulting from appending all its arguments in the order given. It will be truncated to 255 characters if it exceeds that length.

---

## **&DEFAULT: Return string value or default**

**&DEFAULT** (*str-expr*<sub>1</sub>, *str-expr*<sub>2</sub>)

The **&DEFAULT** function takes a string expression as its first argument, *str-expr*<sub>1</sub>. If the value of *str-expr*<sub>1</sub> is not null, **&DEFAULT** returns that value. If the value of *str-expr*<sub>1</sub> is null, **&DEFAULT** returns the value of the string expression *str-expr*<sub>2</sub>. Hence you can use *str-expr*<sub>2</sub> as a default value for the result.

---

## **&GETENV: Return MPW Shell variable value**

**&GETENV** (*str-expr*)

The **&GETENV** function returns the current value of a Macintosh Programmer's Workshop Shell variable specified by the string *str-expr*. If *str-expr* names a variable that is undefined or not exported from the MPW Shell, **&GETENV** returns the null string. MPW Shell variables are explained in the *Macintosh Programmer's Workshop Reference*.

---

## **&INTTOSTR or &I2S: Convert integer to string**

{ **&INTTOSTR** | **&I2S** } (*arith-expr* [ , *width* [ , *hex* ] ] )

The **&INTTOSTR** function converts an integer value to its equivalent string form. The first argument, *arith-expr*, specifies the integer to convert. The second argument, *width*, is optional. It specifies the length of the resulting string. The third argument, *hex*, also optional, and used only if *width* is specified, tells the Macro Processor whether to return a hexadecimal or decimal number. Zero produces decimal conversion; any nonzero value produces hexadecimal conversion. **&INTTOSTR** conforms to these rules:



- If the conversion results in fewer digits than specified by *width*, the left end of the string will be filled with spaces (ASCII \$20) unless you specify a negative value for *width*, in which case the Macro Processor will fill the left end of the string with zero (ASCII \$30) characters.
- If the conversion results in more digits than specified, the Macro Processor will ignore *width* and return a string as long as necessary.
- If you omit the *width* and *hex* arguments, the Macro Processor will assume a *width* of 1 and a *hex* value of 0.

You can also write `&INTTOSTR` as `&I2S`.

---

### **&LOWCASE or &LC: Convert string to lowercase**

`{ &LOWCASE | &LC } (str-expr)`

The `&LOWCASE` function returns the string expression *str-expr* with any uppercase characters in the range A..Z converted to lowercase. You can also write `&LOWCASE` as `&LC`.

---

### **&SETTING: Return directive setting**

`&SETTING (str-expr [, arith-expr])`

The `&SETTING` function determines the current mode setting for the directive named by the string expression *str-expr*. It returns a string whose value is the name of that mode. By using this function, you can access and store the current setting of a directive before temporarily changing it; this lets you restore the setting later to its original value.

The integer expression *arith-expr* is used only when the value of *str-expr* is 'PRINT'—that is, when `&SETTING` is used to return values set by the `PRINT` directive. In this case, *arith-expr* takes a value in the range 0..5 and specifies the nesting level of `PRINT` settings that have been stacked by using the directive's `PUSH` parameter. The default value, 0, refers to the current setting; the maximum value, 5, refers to the setting that held before the last five `PRINT PUSH` directives. This lets you choose up to six different groups of current or prior `PRINT` settings; for example, the directive statement,

```
PRINT      &SETTING('PRINT', 3)
```

restores the settings three levels down on the `PRINT` format stack.

Table 6-2 shows all the directive names that are accepted by `&SETTING` and the possible values that `&SETTING` might return for each directive. In the case of `PRINT`, multiple setting values are concatenated with commas between them.

■ **Table 6-2**      `&SETTING` values

Directive	Possible return values
ALIGN	0, 1
BLANKS	ON, OFF
BRANCH	SHORT, WORD, LONG
CASE	ON, OFF, OBJECT
CODEREFS	FORCEJT, NOFORCEJT, FORCEPC
DATAREFS	ABSOLUTE, RELATIVE
FORWARD	WORD, LONG
MACHINE	MC68000, MC68010, MC68020, MC68030
OPT	ALL, NONE, NOCLR
PRINT	ON, OFF, GEN, NOGEN, PAGE, NOPAGE, WARN, NOWARN, MCALL, NOMCALL, OBJ, NOOBJ, DATA, NODATA, MDIR, NOMDIR, HDR, NOHDR, LITS, NOLITS, STAT, NOSTAT, SYM, NOSYM
STRING	PASCAL, ASIS, C

---

### **&SUBSTR: Return substring of string**

`&SUBSTR(str-expr, start, length)`

The `&SUBSTR` function returns a substring of its first argument, the string expression *str-expr*. Its second and third arguments, *start* and *length*, must be integer expressions. *Start* specifies the position in *str-expr* of the first character of the string returned by `&SUBSTR`; *length* specifies its length. The first character of *str-expr* is position 1. The `&SUBSTR` function follows these rules:

- If the value of *start* is less than one, `&SUBSTR` returns the null string.
- The value of *length* may not be negative.
- If *start* is greater than the length of *str-expr*, `&SUBSTR` returns the null string.
- If *length* is greater than the number of characters remaining in *str-expr* after *start*, `&SUBSTR` will return only the remainder of *str-expr*.

The following are examples of the use of `&SUBSTR`:

Expression	Result
<code>&amp;SUBSTR('foobar', 4, 3)</code>	bar
<code>&amp;SUBSTR('abcdef', 10, 2)</code>	null
<code>&amp;SUBSTR('abcd', 3, 3)</code>	cd

---

### **&TRIM: Trim spaces and tabs from string**

`&TRIM ( str-expr [ , trim-left ] )`

The `&TRIM` function removes leading and trailing spaces or tabs (blanks) from the value of the string expression *str-expr*. The argument *trim-left* must be an arithmetic expression. It gives you these possible actions:

- If you omit *trim-left*, `&TRIM` deletes both leading and trailing blanks.
- If *trim-left* is present and its value is 0, `&TRIM` deletes only trailing blanks.
- If *trim-left* is present and its value is not 0, `&TRIM` deletes only leading blanks.

---

### **&TYPE: Determine identifier type**

`&TYPE ( str-expr )`

The `&TYPE` function determines the type of a macro or nonmacro identifier, returning a string whose value indicates the type. The identifier is the value of the string expression *str-expr*.

- ◆ *Note:* If a macro identifier is the same as a nonmacro identifier, `&TYPE` will return the type of the nonmacro identifier.

The type specification string returned by `&TYPE` may have various values. The possibilities follow these conventions:

- Words enclosed in brackets may or may not be present, depending on the specific type of *str-expr*.
- Words separated by vertical bars (|) represent choices, one of which is always present in the value returned by `&TYPE`.
- The expression *type* represents a template identifier if *str-expr* refers to an object given a type by a template.

- The expression `'dim'` represents the integer value of an array dimension enclosed in brackets.
- The expression `'type'` represents the string value of a type identifier enclosed in parentheses.
- UNDEFINED is the word that `&TYPE` returns if *str-expr* specifies an invalid or undefined identifier.

If *str-expr* specifies a macro variable identifier, `&TYPE` returns one of the following type descriptions:

```
UNDEFINED
PASM [ STRUCTURED ] { INT | STR }
{ SETA | SETC } [ ARRAY ['dim'] ]
MACRO [ { FUNCTION | SYSVAR } ]
```

If *str-expr* specifies a nonmacro identifier, then the identifier is interpreted exactly as if it were used in an absolute expression. It is subject to the standard scope rules given in Chapter 2 in "Scope of Definitions." It may be a fully qualified field reference or a partially qualified reference covered by a `WITH` directive. With nonmacro identifiers, `&TYPE` returns one of the following type descriptions:

```
UNDEFINED
{ CODE | DATA } IMPORT
REG { An | Dn | ZAn | ZDn | CCR | SR | USP | MSP | SFC | DFC | CAAR |
    VBR | CACR | ISP | CRP | SRP | DRP | TC | PSR | PCSR | AC | CAL |
    SCC | VAL | BADn | BACn }
FPREG { FFn | FPCR | FPSR | FPIAR }
RLIST
FRLIST
FCRLIST
{ CODE | DATA } MODULE { EXPORT | ENTRY | IMPORT } [MAIN] ['('type ')']
TEMPLATE [DATA IMPORT] ['('type ')']
TEMPLATE FIELD ['('type ')']
DATA FIELD [ { EXPORT | ENTRY | IMPORT } ] ['('type ')']
{ CODE | DATA } LABEL [ { EXPORT | ENTRY | IMPORT } ]
    [MAIN] ['('type ')']
SET
EQU
OPWORD
```

---

## **&UPCASE or &UC: Convert string to uppercase**

```
{ &UPCASE | &UC } (str-expr)
```

The `&UPCASE` function returns *str-expr* with any lowercase characters in the range a..z converted to uppercase. You can also write `&UPCASE` as `&UC`.

---

## SET array variables

You can define a SET variable as an array, thereby referring to many values with a single identifier. The individual values in a SET array variable are called **elements**. SET array elements may contain either integer or string values and may have either local or global scope. To access an element in a SET array variable, follow the SET variable identifier immediately with a subscript enclosed in brackets.

SET array variables are subject to these rules:

- All elements in a variable must have the same type and scope.
- The subscript may be any integer expression, as long as its value is not 0 or negative.
- You may not use a subscript with a nonarray SET variable.
- A SET array element cannot be accessed without a subscript.

The following are examples of valid SET array element references:

```
&foo[12]
&foo[&bar]
&foo[12+&bar]
&foo[&NBR(&bar)]
```

The following are examples of erroneous set array element references:

```
&foo           No subscript (a good nonarray variable reference)
[12]          No SET variable identifier
&foo [12]     Space between identifier and subscript
```

You may not continue the subscript part of a SET array variable reference to the next line.

---

## Defining SET array variables

You define SET array variables the same way as other SET variables, except that you enclose a decimal, binary, or hexadecimal number in brackets immediately following the variable's identifier in the variable list of your `LCLA`, `LCLC`, `GBLA`, or `GBLC` directive. This number is called the **dimension**; it specifies the number of elements in the array. A SET array variable may not have more than one dimension number. The maximum dimension number allowed is (decimal) 4096. Arrays with dimensions larger than 250 are represented as sparse arrays, and do not necessarily have memory allocated for unassigned elements. You may mix SET array variable definitions with nonarray definitions in the same directive.

If you define a SET array variable as global, you must specify the same dimension number every time you define it again as global. Alternately, you can write an asterisk (\*) in place of the dimension number after the first definition; this indicates that the first dimension number is to be used. The first time a SET array variable is defined, the Macro Processor assigns to each of its elements either 0 or the null string, depending on whether the variable's type is integer or string.

Following are examples of valid definitions of SET array variables:

```
LCLA      &foo[10]
LCLC      &bar[100],&car[54]
GBLA      &totals[4]
GBLC      &currline,&lines[100],&charset[256]
```

---

## Using SET array variables

Once you have defined a SET array variable, you may assign values to its elements with SETA or SETC directives. You can then access these values by referring to the array's elements in subsequent statements or directives. The subscripts in references to SET array elements may be arithmetic expressions of any complexity, as well as absolute integers.

The following example illustrates the definition and use of a SET array variable:

```
MACRO
ArgScan                                ; No parameters, uses &&SYSLIST
LCLC      &a[10]
LCLA      &i,&n
&n        SETA      &NBR(&SYSLIST)    ; Number of actual
                                                ; macro parameters
&i        SETA      1
          WHILE     &i<=&n DO
&a[&i]    SETC      &SYSLIST[&i]
&i        SETA      &i+1
          ENDW
.* the macro parameters have now been stored in the &a
.* array, where they can be manipulated or modified.
.* The macro can always access their original values
.* via &SYSLIST
          - - -                                ; Body of macro
          ENDM
```

---

## Accessing substrings in SET array string elements

When referring to a SET array element that is a string, you may append a second subscript to access individual characters or substrings in the string.

"SETC and String Expressions," given earlier in this chapter, explains how a subscript on a nonarray SET variable of type string lets you access parts of the string value. In the same way, a second subscript on a SET array variable with string elements extracts part of the array string specified by the first subscript.

There are two possible forms for a subscript that accesses part of a string:

'[*array-sub*, *arith-expr*]'      Accesses a single character in the string  
'[*array-sub*, *start*:*length*]'      Accesses a substring in the string

*Array-sub* is the array subscript. *Arith-expr*, *start*, and *length* may be integer expressions of any complexity. *Arith-expr* accesses a single character, counting the first character of the string as 1. *Start* specifies the position of the first character of a substring, and *length* specifies its length.

Here are some examples of accessing parts of a SET array element with string value; they all assume that the value of the sixth element of the SET array variable `&foo[6]` is the string 'slangword':

Reference	Value
<code>&amp;foo[6,4]</code>	'n'
<code>&amp;foo[6,5:3]</code>	'gwo'
<code>&amp;foo[x+3,x:4]</code>	'angw' if $x = 3$

The use of a second subscript with a SET string array variable identifier is subject to these rules:

- When accessing a single character, if the value of *arith-expr* is greater than the length of the string, the Macro Processor will return a string consisting of a single space character (ASCII \$20).
- When accessing a substring, if *start* is 0 or is greater than the length of the string, the Macro Processor will return the null string.
- When accessing a substring, if the position of *start* is within the string but fewer than *length* characters remain, the Macro Processor will return only the remainder of the string.

---

## Assembler system variables

Assembler system variables are variables which are defined and assigned values by the Assembler. These values may change during the assembly process. If you define them in your source text, you will override their original purpose. They are also subject to the restrictions given for SET variables earlier in this chapter. Table 6-3 lists the Assembler system variables.

■ **Table 6-3** Assembler system variables

---

Name	Value	Type
&SYSNDX	Index number of the current macro call	Integer or string
&SYSLIST	Current macro parameter list	String
&SYSSEG	Identifier of the current code segment	String
&SYSMOD	Identifier of the current code module	String
&SYSDATE	Current date	String
&SYSTEME	Current time	String
&SYSTOKEN	Token code from &LEX call	Integer
&SYSTOKSTR	Token string from &LEX call	String
&SYSVALUE	Symbol value returned by &FINDSYM	Integer or string
&SYSFLAGS	Symbol flags returned by &FINDSYM	Integer
&SYSLOCAL	ID of the Assembler's local symbol table	Integer
&SYSGLOBAL	ID of the Assembler's global symbol table	Integer

---

### **&SYSINDEX or &SYSNDX: Macro call index**

The Macro Processor assigns &SYSINDEX the four-digit number 0001 when it encounters the first macro call directive in your source text. It increases this number by one at each subsequent macro call directive, including inner macro calls. The value of &SYSINDEX remains the same throughout macro expansion (being restored after inner macro calls). You can also write &SYSINDEX as &SYSNDX.

If &SYSINDEX is used in a model statement or in a string expression, the Macro Processor treats it as a string symbol. Its value will include any leading zeros needed to create a four-digit number. If &SYSINDEX is used in an integer expression (for example, in the operand of a SETA directive), the Macro Processor treats it as an ordinary integer.



A typical use of `&SYSINDEX` is to concatenate it with other characters, thereby creating unique labels for statements generated during different expansions of the same macro model statement.

---

### **&SYSLIST or &SYSLST: Macro operand list**

The `&SYSLIST` variable gives you an alternate way to access macro parameters. However, it accesses only the positional parameters; it does not access keyword parameters, described in Chapter 5. You can also write `&SYSLIST` as `&SYSLST`.

You can use `&SYSLIST` only in macro definitions.

`&SYSLIST[n]` refers to the  $n$ th positional operand of the current macro. If the  $n$ th operand is a sublist, `&SYSLIST[n,m]` refers to the  $m$ th sublist element of the  $n$ th operand. You may use any absolute integer expression for the subscripts  $n$  and  $m$ , as long as their values are greater than zero and not greater than the numbers of parameters or sublist elements. If the  $m$ th element of the  $n$ th parameter is itself a sublist, then `&SYSLIST[n,m,k]` refers to its  $k$ th element, and so on.

You can use the `&NBR` function with `&SYSLIST` to determine the number of positional operands that were specified when the current macro was called. When applied to `&SYSLIST[n]`, `&NBR` returns these values:

- the number of elements in the  $n$ th operand, if it is a sublist
- 0 if the  $n$ th operand is not a sublist
- 0 if the  $n$ th operand was omitted in the current macro call

Here are some examples of values returned by `&SYSLIST` expressions in a macro definition whose parameter list is `A, (B,C), D`:

<b>Expression</b>	<b>Value</b>
<code>&amp;SYSLIST[1]</code>	A
<code>&amp;SYSLIST[2]</code>	(B,C)
<code>&amp;SYSLIST[3]</code>	D
<code>&amp;SYSLIST[2,1]</code>	B
<code>&amp;SYSLIST[2,2]</code>	C

---

**&SYSSEG: Current segment identifier**

The value of the `&SYSSEG` variable is the name of the current code segment.

---

**&SYSMOD: Current module identifier**

The value of the `&SYSMOD` variable is the identifier of the current module. This value is the null string if `&SYSMOD` is used outside a module or inside an unnamed module.

---

**&SYSDATE: Current date**

The value of the `&SYSDATE` variable is the current date, expressed in the format *dd-mmm-yy* where

- dd*        -     String containing two-numeral day of the month
- mmm*      -     String containing first three letters of the current month
- yy*        -     String containing two-numeral year

All these values are copied from the Macintosh internal clock.

---

**&SYSTIME: Current time**

The value of the `&SYSTIME` variable is the current time, expressed in the format *hh:mm:ss* where

- hh*        -     String containing two-numeral hour of the day (24-hour basis)
- mm*        -     String containing two-numeral minute
- ss*        -     String containing two-numeral second

All these values are copied from the Macintosh internal clock.

---

**&SYSTOKEN and &SYSTOKSTR: Values set by &LEX**

The values of `&SYSTOKEN` and `&SYSTOKSTR` are set every time `&LEX` is called. `&SYSTOKEN` contains an integer code indicating the token type, and `&SYSTOKSTR` contains the token's string value. Table 6-1 shows all possible values of these variables.

---

**&SYSVALUE and &SYSFLAGS: Values set by &FINDSYM**

The values of `&SYSVALUE` and `&SYSFLAGS` are set every time `&FINDSYM` is called. `&SYSVALUE` contains the value associated with the symbol found, and `&SYSFLAGS` contains a positive 16-bit integer that indicates certain characteristics of the symbol. These variables have different types and values, depending on how `&FINDSYM` was called. For further information, see "`&FINDSYM`" earlier in this chapter.

---

**&SYSLOCAL and &SYSGLOBAL: System symbol table ID's**

These variables have fixed values assigned by the Assembler. They are used only as parameters for the `&ENTERSYM` and `&FINDSYM` functions. `&SYSLOCAL` may be used only in source text inside a code or data module. For further information, see "`&ENTERSYM`" and "`&FINDSYM`" earlier in this chapter.



## Chapter 7 Macro and Conditional-Assembly Directives

THE POWER OF THE MACRO LANGUAGE DEPENDS PRIMARILY on its ability to loop and branch, thereby letting you program the ways that it expands your original source text. This chapter discusses the following directives:

GOTO	Unconditional jump to another part of the source text
IF...GOTO	Conditional jump to another part of the source text
IF...ENDIF	Conditional assembly of enclosed source text
ELSEIF, ELSE	Further conditions on assembly of enclosed source text
WHILE...ENDWHILE	Conditional looping of enclosed source text
ACTR	Set maximum number of branches in macro expansion
EXITM	Unconditional termination of macro
WRITE, WRITELN	Write information to the diagnostic output
AERROR	Generate an Assembler error
ANOP	Assembler NOP

Many of these directives use Boolean expressions for control. Such expressions are discussed in the next section. ■

### *Contents*

Boolean control expressions	175
Comparing two integer expressions	175
Comparing two string expressions	175
Comparing integer and string expressions	176
GOTO, IF...GOTO, and macro labels: Branching	176
IF, ELSEIF, ELSE, and ENDIF: Conditional assembly	178
WHILE and ENDWHILE: Looping	179
CYCLE and LEAVE directives	180
ACTR: Limit looping	180
EXITM or MEXIT: Exit macro	181
WRITE and WRITELN: Write to diagnostic output file	181
AERROR: Error generation	182
ANOP: Assembler NOP	182



---

## Boolean control expressions

Conditional-assembly directives are controlled by Boolean expressions, in which integer values are treated as true or false. Hence every absolute integer expression may also be used as a Boolean expression. The Macro Processor interprets an integer value of 0 as a Boolean value of false; it interprets any nonzero integer value as a Boolean value of true. For correct results with expressions containing AND or OR, however, you should always use 1 to denote the Boolean value true.

In addition, however, Boolean expressions may be constructed from **logical** and **comparison operators**. To extend the utility of these operators in macro constructs, the comparison operators may be used for string comparisons as well as for integer comparisons.

The logical operators are OR, XOR, AND, and NOT. The comparison operators are =, <> or ≠, >, <, >= or ≥, and <= or ≤. The syntax rules for writing these operators are discussed more fully in "Expressions" in Chapter 2.

Each operand of a comparison expression may have either an integer or a string value. The ways the Macro Processor evaluates the whole expression depends on the types of the two operands, as discussed here.

---

### Comparing two integer expressions

The Macro Processor performs integer comparisons in the normal arithmetic way; the results are always either 0 (false) or 1 (true).

---

### Comparing two string expressions

String comparisons compare the values of two string expressions for equality, inequality, and relative alphabetical ordering. The results are integer values—0 for `false`, 1 for `true`.

The Macro Processor distinguishes between uppercase and lowercase when comparing strings. You can mask the distinction by using the `&UPCASE` or `&LOWCASE` functions described in "SETC and String Expressions" in Chapter 6.

Two strings are **equal** if they are indistinguishable; otherwise they are **unequal**.

The Macro Processor ranks strings by **relative ASCII ordering** for comparisons using the  $>$ ,  $<$ ,  $\geq$ , and  $\leq$  operators. It performs this ranking by the following steps:

1. The two strings are compared a character at a time, starting with the first character.
2. Two corresponding characters are compared. If the ASCII value of one character is greater than the other, then the corresponding string is greater than the other.
3. If the two corresponding characters are equal, the point of comparison advances to the next character in each string, and the process returns to step 2.
4. If the end of one string is reached before the end of the other, its value is less than the other string.
5. If the ends of both strings have been reached, the two strings are equal.

---

### Comparing integer and string expressions

To compare an integer with a string, the Macro Processor first converts the second operand to the type of the first operand. If the first operand is type integer, the second operand will be converted to an integer by interpreting the string contents as an integer constant. It does this by following the same parsing rules as `&ISINT`, described in "SETA and String Expressions" in Chapter 6. If the string does not contain an integer, the Assembler reports an error. If the first operand is a string, the integer operand will be converted to a string containing the digits that represent the operand's value. Note that comparisons of the string representations of integers do not always yield the same results as arithmetic comparison of the same integers.

---

### GOTO, IF...GOTO, and macro labels: Branching

```
[macro-label]    GOTO    [+ | -]{ macro-label | str-expr }  
[macro-label]    IF      bool-expr GOTO [+ | -]{ macro-label | str-expr }
```

You can use the `GOTO` and conditional `GOTO` directives to alter the sequence in which the Assembler processes your source text. The destination of a `GOTO` directive is always a **macro label**. You can specify a macro label by using either a macro label identifier or an expression whose value is a string containing a macro label identifier.



A macro label consists of a period followed by a valid identifier, and is defined when it appears in the label field of a statement or directive. If the period is not the first character of the source text line, you must terminate the macro label with a colon. You may write macro labels in any machine instruction statement, or in any directive statement that accepts a macro label. Macro labels conform to these rules:

- Macro labels defined within a macro are local to that macro.
- Macro labels with the same identifier are not allowed in the same scope.
- A macro label may appear on a line by itself, without an operation. This is necessary in cases where the destination line already has a label. Alternatively, you can write a label on an `ANOP` directive.
- The Macro Processor does not distinguish between uppercase and lowercase when interpreting macro labels.

If the jump is within a macro, you may optionally write either a plus or a minus character before the macro label to tell the Macro Processor whether the jump is forward or backward. The label must, of course, be located where the plus or minus indicates it will be. Using the plus or minus decreases assembly time.

The `GOTO` directive causes assembly processing to jump unconditionally to the source text line where the specified label is defined. Any lines passed over are not processed.

The `IF...GOTO` directive determines the value of the expression *bool-expr*. If it is true (nonzero), the assembly process jumps to the source text line where the specified label is defined. If it is false (zero), processing continues with the succeeding source text line as usual.

When using `GOTO` and `IF...GOTO`, bear these rules in mind:

- `GOTO` and `IF...GOTO` directives outside macros may jump only to macro labels that are defined in subsequent statements; backward jumps are not allowed outside macros.
- `GOTO` and `IF...GOTO` directives in a macro definition may not cause a branch to any label outside the body of that macro except to the macro label (if any) in the macro's `ENDM` statement.

The following example illustrates the use of `GOTO` directives and macro labels. It uses a conditional `GOTO` directive to avoid the unnecessary code sometimes generated by the sample macro given in Chapter 5 in "Symbolic Parameters."

```

                MACRO
Incr           &src, &dest
IF            UPCASE(&src)='D0' GOTO .lab1
MOVE.W       &src, D0
.lab1        ADDQ.W  #1, D0
IF            UPCASE(&dest)='D0' GOTO .lab2
MOVE.W       D0, &dest
.lab2        ENDM

```

---

## IF, ELSEIF, ELSE, and ENDIF: Conditional assembly

```
[macro-label]      IF bool-expr THEN
                   statements
                   ELSEIF bool-expr: THEN
                   statements
                   ELSE
                   statements
                   ENDIF
```

The IF, ELSEIF, ELSE, and ENDIF directives provide facilities for conditional assembly. You use them to enclose sections of source text that the Assembler will process only if certain Boolean values are true. You may abbreviate ENDIF as ENDI.

Each IF directive must be followed by an ENDIF directive. Together they form an IF . . . ENDIF construct. ELSEIF and ELSE directives are optional, but when they appear they must be located within an IF . . . ENDIF construct, as indicated in the above syntax diagram. Here are the rules:

- You can write any number of ELSEIF directives, but only one ELSE directive.
- The ELSE directive, if any, must follow all the ELSEIF directives.
- The region of source text controlled by IF extends to its corresponding ENDIF, or to the first ELSEIF or ELSE, whichever comes first.
- The region of source text controlled by each ELSEIF extends to the next ELSEIF, ELSE, or ENDIF, whichever comes first.
- The region of source text controlled by ELSE extends to ENDIF.

When the Macro Processor processes IF and ELSEIF directives, it evaluates their Boolean expressions. For a given IF . . . ENDIF construct, the first Boolean value of true in an IF or ELSEIF directive will cause assembly of the source text controlled by that directive. All further ELSEIF directives in the construct, and the source text they control, will be skipped. If the Macro Processor does not find any true IF or ELSEIF directives in the construct, it assembles the source text controlled by ELSE.

Remember that an entire IF . . . ENDIF construct may be skipped if it is enclosed in an IF . . . ENDIF or WHILE . . . ENDWHILE construct whose Boolean control value is false.

You can nest IF . . . ENDIF constructs; the Macro Processor will associate the last IF with the first ENDIF, the next-to-last IF with the second ENDIF, and so on.

The following example illustrates the use of an IF . . . ENDIF construct. It improves the example given in Chapter 5 in "Macro Comments" by generating a debug procedure header only if the global SET variable &Debugging is true (nonzero).

```
MACRO
  DbgHead
  GBLA      &Debugging      ; Declare (import)
                                ; global variable
  IF        &Debugging THEN ; Header only if debugging
  LINK      A6, #0          ; Debug header
  ENDIF
  ENDM

[macro-label]      WHILE bool-expr DO
                    statements
[macro-label]      ENDW[HILE]
```

---

## WHILE and ENDWHILE: Looping

The WHILE and ENDWHILE directives provide conditional looping. You can use them only in macro definitions. You use them in pairs to enclose sections of source text that the Assembler will process repeatedly, as long as the value of *bool-expr* is true. The result is called a **WHILE . . . ENDWHILE construct**. You may abbreviate ENDWHILE as ENDW.

When the Macro Processor encounters a WHILE directive, it evaluates its Boolean expression. If the value of the expression is true (nonzero), the Macro Processor processes source text until the corresponding ENDWHILE. If the Boolean value is false (0), the Macro Processor skips to the source text line immediately following ENDWHILE.

When the Macro Processor encounters ENDWHILE (which happens only if the corresponding WHILE directive was true), the next line it processes will be the corresponding WHILE directive. In other words, it loops back from ENDWHILE to WHILE. It evaluates the WHILE directive's Boolean expression again and repeats the procedure just described.

WHILE . . . ENDWHILE constructs may be nested; the Macro Processor will associate the last WHILE with the first ENDWHILE, the next-to-last WHILE with the second ENDWHILE, and so on.

---

## CYCLE and LEAVE directives

The `CYCLE` and `LEAVE` directives are used with `WHILE` statements. The `CYCLE` directive makes the next executed statement the enclosing `WHILE` directive. The `LEAVE` directive makes the next executed statement the corresponding `ENDWHILE` directive.

Here is the syntax of the `CYCLE` and `LEAVE` directives:

```
[macro-label] CYCLE [ ( macro-label | str-expr ) ]  
[macro-label] LEAVE [ ( macro-label | str-expr ) ]
```

You can specify macro label (or a string expression evaluating to a macro label) as an argument to a `CYCLE` or `LEAVE` directive in order to identify one of several enclosing `WHILE` directives. (This assumes that each of the enclosing `WHILE` directives was labeled with the appropriate macro label.) Here is an example:

```
.lab WHILE...DO  
    ...  
    WHILE...DO  
        ...  
        IF...THEN  
            LEAVE.lab           ;will leave outer WHILE  
        ENDIF  
    ...  
    ENDWHILE  
    ...  
    ENDWHILE
```

---

## ACTR: Limit looping

```
[macro-label] ACTR arith-expr
```

The `ACTR` directive sets the maximum number of `GOTO`, `IF . . . GOTO`, and `WHILE` branches that the Macro Processor can make. This number is the value of *arith-expr*. If you write an `ACTR` directive in the body of a macro definition, the limit applies to each expansion of that macro. If you write it anywhere else, it applies to the whole assembly process except macro expansions. The limit remains in force until the end of the assembly process or until it is superseded by another `ACTR` directive. In the absence of any `ACTR` directive, the preset limit value is 512.

The `ACTR` directive is useful to prevent macros that have not been debugged from becoming processed in infinite loops.

Here are some rules about ACTR:

- The value of *arith-expr* must be positive.
- When counting branches during macro expansion, ACTR ignores branches made during inner macro calls.
- Any time the actual number of branches exceeds the limit set by ACTR, the Assembler reports an error.
- If the limit set by ACTR is exceeded during a macro expansion, that macro expansion will terminate.
- If the limit set by ACTR is exceeded outside any macro expansion (which could be caused only by forward GOTO branches), the entire assembly process will terminate.

---

## EXITM or MEXIT: Exit macro

*[macro-label]* (EXITM | MEXIT)

The EXITM directive terminates expansion of the macro currently being called. You can use EXITM only within the body of a macro definition. You can also write EXITM as MEXIT.

After an EXITM directive is processed, the next source text line to be assembled is the one just after the macro call that invoked the macro just terminated. If that macro call is an inner macro call (nested inside another macro) then the Macro Processor continues processing the higher-level macro.

- ◆ *Note:* Don't confuse EXITM with ENDM, which is the last directive in every macro definition.

---

## WRITE and WRITELN: Write to diagnostic output file

*[macro-label]* WRITE *item*,...  
*[macro-label]* WRITELN [*item*],...

The WRITE and WRITELN directives write information to the diagnostic output file during the assembly process. You may use them both within and outside macros.

The *item* operands may be symbolic parameters, SET variables, string expressions, or integer expressions. The Assembler will send their values to the diagnostic output file with no intervening spaces. The difference between WRITE and WRITELN is that WRITELN sends a final return character (ASCII \$0D) after the last item. WRITELN with no operand sends a single return character to the diagnostic output.

WRITE and WRITELN are typically used for debugging macros or for displaying information about the current state of the assembly process.

---

## AERROR: Error generation

*{macro-label}*      AERROR      *str-expr*

The AERROR directive generates a synthetic Assembler error and sends the value of the string expression *str-expr* to the diagnostic output as the error message. You can use AERROR both within macro definitions and outside macros.

AERROR is typically used in conjunction with a conditional directive to report when conditions required for a successful assembly or macro expansion have not been met. Here is an example:

```
AERROR            'Bad parameter value to macro foo'
```

---

## ANOP: Assembler NOP

*{macro-label}*      ANOP

The ANOP directive lets you specify a macro label as the destination of a GOTO directive. The macro-label destination of a GOTO directive must be placed in the label field of the statement to which it branches. If there is already a symbol in the label field (for example, a SETA assignment) you cannot put the macro label there. However, you can achieve the same effect by labeling an ANOP statement just before the desired destination statement.

- ◆ *Note:* Using ANOP is not the quite the same as defining a macro label on a line with a blank operation field. ANOP does not generate a line from the macro, while a macro label on a line by itself will generate a blank line. This difference shows up on an assembly listing.

Part III **Appendixes**





## Appendix A Generic Instruction Formats

YOU CAN WRITE CERTAIN ASSEMBLY-LANGUAGE INSTRUCTIONS IN generic form, letting the Assembler transform them automatically to the final instructions that will appear in your source text. There are three general reasons for writing generic instructions:

- **Optimization:** The Assembler transforms your instructions if they can be encoded more efficiently. The result occupies less memory and often runs faster as well.
- **Convenience:** The Assembler transforms your instructions on the basis of their context, to make coding easier and your source text more readable.
- **Compatibility:** The Assembler transforms your instructions to make them compatible with other assemblers. Other assemblers are discussed in Appendix D.

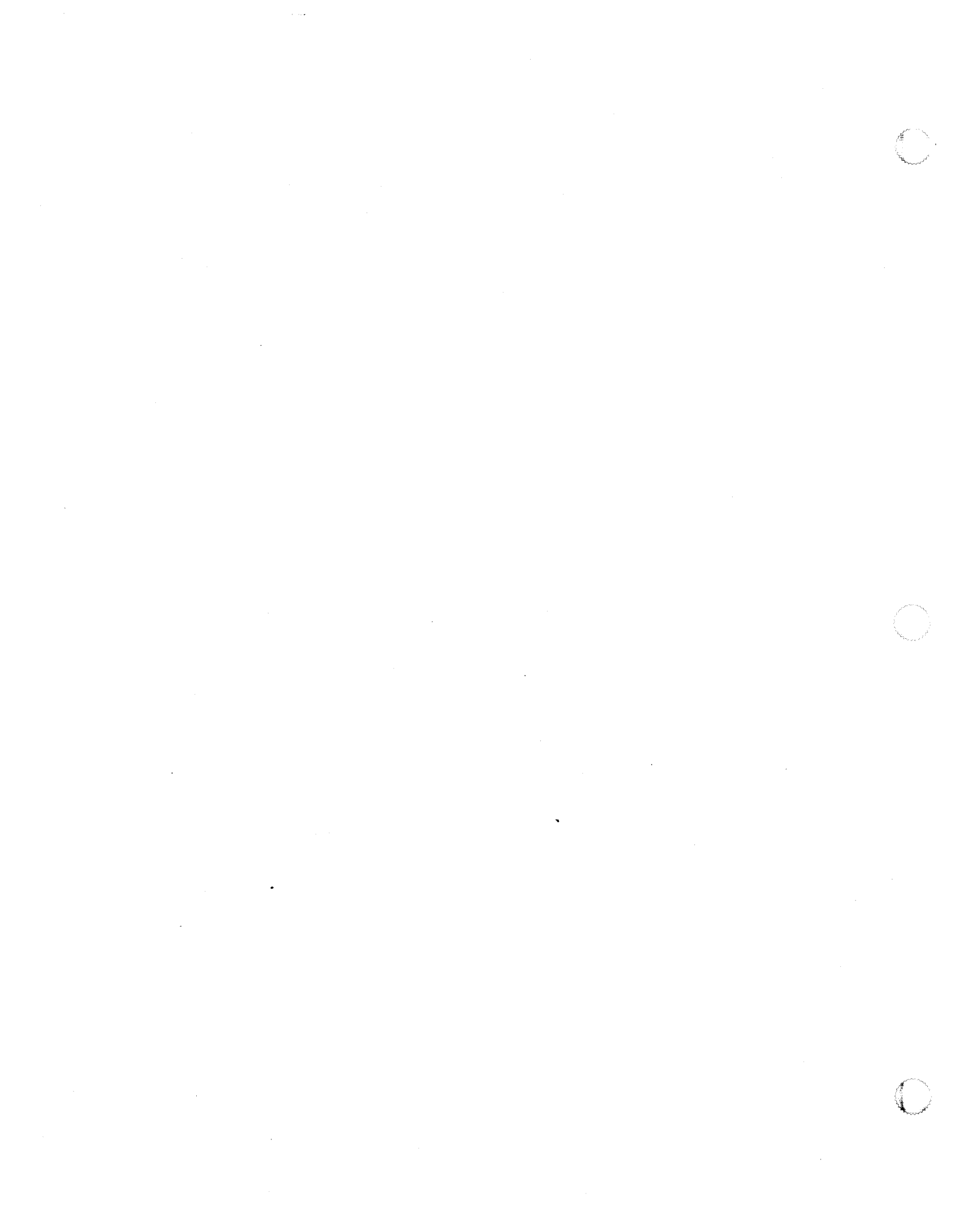
The `OPT` directive, described in Chapter 4 in “Assembly Options,” gives you some control of whether or not the Assembler performs address optimization and generic instruction substitution.

Whenever the Assembler performs a generic instruction substitution, it identifies that statement in the Assembly listing with a letter *G* in the flags column. See Appendix C for a full description of the assembly listing format.

For each instruction that you can write generically, Table A-1 lists the following:

- the generic form that you write
- the form generated by the Assembler
- any conditions that must be met for the Assembler to perform the transformation

The table is grouped by the reason for the transformation. ■



■ Table A-1 Generic instruction conversions

Generic Instruction		Assembled form	Conditions	
<b>Optimization</b>				
ADDA	#data, <ea1>	ADDQ	#data, <ea1>	#data = 1..8
CLR.L	Dn	MOVEQ	#0, Dn	
MOVE	#0, <ea3>	CLR	<ea3>	<ea3> ≠ Dn
MOVE.L	#data, An	MOVEA.W	#data, An	#data = -32768..32767
MOVE.L	#data, Dn	MOVEQ	#data, Dn	#data = -128..127
MOVE	#0, An	SUBA	An, An	
MOVEA	#0, An	SUBA	An, An	
MOVEA.L	#data, An	MOVEA.W	#data, An	#data = 0..32767
ADDA.L	#data, An	MOVEA.W	#data, An	#data = 0..32767
CMPA.L	#data, An	MOVEA.W	#data, An	#data = 0..32767
SUBA.L	#data, An	MOVEA.W	#data, An	#data = 0..32767
MOVEQ	#0, An	SUBA	An, An	
SUB	#data, <ea1>	SUBQ	#data, <ea1>	#data = 1..8
SUBA	#data, <ea1>	SUBQ	#data, <ea1>	#data = 1..8
<b>Convenience</b>				
ADD	#data, <ea3>	ADDI	#data, <ea3>	
ADD	<ea0>, An	ADDA	<ea0>, An	
AND	#data, <ea3>	ANDI	#data, <ea3>	
BNZ	label	BNE	label	
Bcc.S	label	NOP	label	label = next instruction
BT	label	BRA	label	
BZ	label	BEQ	label	
CMP	#data, <ea3>	CMPI	#data, <ea3>	
CMP	<ea3>, An	CMPA	<ea3>, An	
CMP	(An)+, (An)+	CMPM	(An)+, (An)+	
DBNZ	Dn, label	DBNE	Dn, label	
DBZ	Dn, label	DBEQ	Dn, label	
DBRA	Dn, label	DBF	Dn, label	
EOR	#data, <ea3>	EORI	#data, <ea3>	
MOVE	<ea0>, An	MOVEA	<ea0>, An	
MOVEM	<ea8>, An	MOVEA	<ea8>, An	
OR	#data, <ea3>	ORI	#data, <ea3>	

(continued)

■ Table A-1 (continued)

Generic instruction conversions

Generic instruction		Assembled form		Conditions
<b>Convenience, (continued)</b>				
SNZ	<ea3>	SNE	<ea3>	
SUB	#data, <ea3>	SUBI	#data, <ea3>	
SUB	<ea0>, An	SUBA	<ea0>, An	
TNZ		TNE		
TZ		TEQ		
TPNZ	#data	TPNE	#data	
TPZ	#data	TPEQ	#data	
<b>Compatibility</b>				
BHS	label	BCC	label	
BLO	label	BCS	label	
DBHS	Dn, label	DBCC	Dn, label	
DBLO	Dn, label	DBCS	Dn, label	
EXTB	Dn	EXT.W	Dn	
EXTW	Dn	EXT.L	Dn	
SHS	<ea3>	SCC	<ea3>	
SLO	<ea3>	SCS	<ea3>	
THS		TCC		
TLO		TCS		
TPHS	#data	TPCC	#data	
TPLO	#data	TPCS	#data	

## Appendix B Syntax Diagrams

THIS APPENDIX GROUPS TOGETHER ALL THE SYNTAX DIAGRAMS USED IN THIS BOOK. The listing is divided into the following sections:

- assembly-language addresses (described in Chapter 3)
- special address formats (described in Chapter 3)
- general assembly directives (described in Chapter 4)
- macro and SET variable directives (described in Chapter 5)
- SET variable functions (described in Chapter 5)

For an explanation of how to interpret these diagrams, see “Notation Conventions” in the Preface. ■

### **Contents**

Assembly-language addresses	191
Addressing modes	191
Address optimizations	192
Special address formats	192
MC68000 instructions	192
MOVEM: Multiple moves	192
MC68020 instructions	192
MULS and MULU: Signed and unsigned multiplication	192
DIVS and DIVU: Signed and unsigned division	193
TDIVS and TDIVU: Truncated signed and unsigned division	193
PACK and UNPK: Packing and unpacking	193
CAS and CAS2: Comparing and swapping	193
Bit field instructions	193
Tcc and TPcc: Trap on condition	193

MC68881 and MC68882 instructions 194  
    FMOVEM with explicit register lists 194  
    FMOVE with packed BCD data 194  
    FSINCOS: Simultaneous sine and cosine 194  
    FTcc and FTPcc: Floating-point trap on condition 194  
    FTEST: Test operand and set floating-point condition codes 194  
MC68851 instructions 195  
Literals 195  
General assembly directives 196  
Macro and SET variable directives 200  
SET variable functions 202

## Assembly-language addresses

For an explanation of the symbols used in address syntax diagrams, see Table 3-1.

### Addressing modes

Here is a short list of the addressing modes of the MC68xxx and the addressing optimizations performed by the Assembler.

Mode	Addressing mode	Effective address syntax
0	Data register direct	$Dn$
1	Address register direct	$An$
2	Address register indirect	$(An)$
3	Postincrement register indirect	$(An)+$
4	Predecrement register indirect	$-(An)$
5	Indirect with 16-bit displacement	$d(An)$
6	Indirect with indexing plus 8-bit displacement	$d(An, Xn)$
6*	Indirect with indexing plus base displacement	$(bd, An, Xn*s) \quad bd(An, Xn*s)$
6*	Indirect with preindexing	$([bd, An, Xn*s], od)$
6*	Indirect with postindexing	$([bd, An], Xn*s, od)$
70	Absolute word (16 bits)	$ae \quad (ae).W$
71	Absolute long (32 bits)	$ae \quad (ae).L$
72	PC-relative with 16-bit displacement	$re \quad d(PC)$
73	PC-relative, indexing, 8-bit displacement $d(Dn)$	$d(PC, Xn)$
73*	PC-relative, indexing, base displacement $(bd, PC, Xn*s)$	$bd(PC, Xn*s)$
73*	PC-relative, indexing, base displacement $bd(Xn*s)$	$bd(Xn*s)$
73*	PC-relative with preindexing	$([bd, PC, Xn*s], od)$
73*	PC-relative with postindexing	$([bd, PC], Xn*s, od)$
74	Immediate	$*ae$
72	Literal (PC-relative with 16-bit displacement)	$*ae \quad *(ae).W*(ae).L$

\* Modes usable only with the MC68020 and MC68030

## Address optimizations

Original form	Condition for optimization	Optimized form
$(bd, An, Xn^*s)$	Size of $bd \leq 8$ bits	$bd(An, Xn^*s)$
$(An, Xn^*s)$	Omitted $bd$ ( $bd = 0$ )	$0(An, Xn^*s)$
$(bd, PC, Xn^*s)$	Size of $bd \leq 8$ bits	$bd(PC, Xn^*s)$
$(bd, An)$	Size of $bd \leq 16$ bits	$bd(An)$
$(bd, PC)$	Size of $bd \leq 16$ bits	$bd(PC)$
$d(An)$	$d = 0$ $(An)$	

---

## Special address formats

Some instructions accept address formats that are unusual or are special cases of more usual formats. These instructions are described here.

---

## MC68000 instructions

### MOVEM: Multiple moves

MOVEM .size	<i>rlist, ea</i>
MOVEM .size	<i>ea, rlist</i>
size ::= W   L	

---

## MC68020 instructions

### MULS and MULU: Signed and unsigned multiplication

MULS .L	<i>ea, Dl</i>	32 x 32 → 32
MULS .L	<i>ea, Dh:Dl</i>	32 x 32 → 64
MULU .L	<i>ea, Dl</i>	32 x 32 → 32
MULU .L	<i>ea, Dh:Dl</i>	32 x 32 → 64



### DIVS and DIVU: Signed and unsigned division

DIVS.L	<i>ea, Dq</i>	32/32 → 32q
DIVS.L	<i>ea, Dr:Dq</i>	64/32 → 32r:32q
DIVU.L	<i>ea, Dq</i>	32/32 → 32q
DIVU.L	<i>ea, Dr:Dq</i>	64/32 → 32r:32q

### TDIVS and TDIVU: Truncated signed and unsigned division

TDIVS.L	<i>ea, Dq</i>	32/32 → 32q
TDIVS.L	<i>ea, Dr:Dq</i>	32/32 → 32r:32q
TDIVU.L	<i>ea, Dq</i>	32/32 → 32q
TDIVU.L	<i>ea, Dr:Dq</i>	32/32 → 32r:32q

### PACK and UNPK: Packing and unpacking

PACK	<i>-(Ax), -(Ay), #adjustment</i>
PACK	<i>Dx, Dy, #adjustment</i>
UNPK	<i>-(Ax), -(Ay), #adjustment</i>
UNPK	<i>Dx, Dy, #adjustment</i>

### CAS and CAS2: Comparing and swapping

CAS.size	<i>Dc, Du, ea</i>
CAS2.size	<i>Dc1:Dc2, Du1:Du2, (Rn1):(Rn2)</i>
size ::= B   W   L	

### Bit field instructions

BFCHG	<i>ea { 'offset: width' }</i>
BFCLR	<i>ea { 'offset: width' }</i>
BFEXTS	<i>ea { 'offset: width' }, Dn</i>
BFEXTU	<i>ea { 'offset: width' }, Dn</i>
BFFFO	<i>ea { 'offset: width' }, Dn</i>
BFINS	<i>Dn, ea { 'offset: width' }</i>
BFSET	<i>ea { 'offset: width' }</i>
BFTST	<i>ea { 'offset: width' }</i>

### Tcc and TPcc: Trap on condition

TCC	
TPCC.size	<i>#ae</i>
size ::= W   L	

---

## MC68881 and MC68882 instructions

### FMOVEM with explicit register lists

```
FMOVEM .size    fp-rlst, ea
FMOVEM .size    ea, fp-rlst
size ::= L | X
```

### FMOVE with packed BCD data

```
FMOVE.P        FPn, ea
FMOVE.P        FPn, ea {'#k'}
FMOVE.P        FPn, ea {'Dn'}
```

### FSINCOS: Simultaneous sine and cosine

```
FSINCOS .size   ea, FPC:FPS
FSINCOS .X      FPm, FPC:FPS
size ::= B | W | L | S | D | X | P
```

### FTcc and FTPcc: Floating-point trap on condition

```
FTCC
FTPcc.size     #ae
size ::= W | L
```

### FTEST: Test operand and set floating-point condition codes

```
FTEST .size    ea
FTEST .X      FPn
size ::= B | W | L | S | D | X | P
```

---

## MC68851 instructions

Opcode	Operand format	Sizes	Notes
PBCC.size	label	W   L	
PDBCC.size	Dn, label	W	
PFLUSH	fc, #ae [, ea]		1
PFLUSHA			
PFLUSHS	fc, #ae [, ea]		1
PFLUSHR	ea	D	4
PLOADR	fc, ea		1
PLOADW	fc, ea		1
PMOVE	PMMU-reg, ea	B   W   L   D	2
PMOVE	ea, PMMU-reg	B   W   L   D	2,4
PRESTORE	ea		
PSAVE	ea		
PSCC	ea	B	
PTESTR	fc, ea, #ae [, An]		1
PTESTW	fc, ea, #ae [, An]		1
PTCC			3
PTPCC	#ae	W   L	3
PVALID	VAL, ea	L	2
PVALID	An, ea	L	

1. *fc* ::= #ae (specified as 4 bits in the command word)
2. *Dn* (contained in the lower 4 bits of *Dn*)
3. *SFC* (contained in the processor's source function register)
4. *DFC* (contained in the processor's destination function code register)

---

## Literals

PEA	#(ae).W	Immediate word data for PEA
PEA	#(ae).L	Immediate long-word data for PEA
PEA	#(ae).S	Immediate single-precision data for PEA
PEA	#(ae).D	Immediate double-precision data for PEA
PEA	#(ae).X	Immediate extended data for PEA
PEA	#(ae).P	Immediate packed BCD data for PEA
LEA	#(ae).W, An	Immediate word data for LEA
LEA	#(ae).L, An	Immediate long-word data for LEA
LEA	#(ae).S, An	Immediate single-precision data for LEA
LEA	#(ae).D, An	Immediate double-precision data for LEA
LEA	#(ae).X, An	Immediate extended data for LEA
LEA	#(ae).P, An	Immediate packed BCD data for LEA

---

## General assembly directives

[ <i>macro-label</i> ]	ALIGN	[ <i>expr</i> ]
[ <i>name</i> ]	BLANKS	{ ON   Y[ES] } { OFF   N[O] }
[ <i>macro-label</i> ]	BRANCH	{ S[HORT]   B[YTE] } { W[ORD] } { L[ONG] }
[ <i>macro-label</i> ]	CASE	{ ON   Y[ES] } { OFF   N[O] } { OBJ[ECT] }
[ <i>macro-label</i> ]	CODE	
[ <i>macro-label</i> ]	CODEREFS	{ F[ORCE][JT] } { NOF[ORCE][JT] } { F[ORCE]PC }
[ <i>macro-label</i> ]	COMMENT	<i>str-expr</i>
[ <i>macro-label</i> ]	DATA	{ { INCR[EMENT] } } { { DECR[EMENT] } } { MAIN }

[ <i>macro-label</i> ]	DATAREFS	$\left\{ \begin{array}{l} R[ELATIVE] \\ A[ABSOLUTE] \end{array} \right\}$
[[ <i>label</i>   <i>macro-label</i> ]]	DC[.size]	{ <i>expr</i>   <i>string</i> },...
[[ <i>label</i>   <i>macro-label</i> ]]	DCB[.size]	<i>length</i> , { <i>expr</i>   <i>string</i> }
[[ <i>label</i>   <i>macro-label</i> ]]	DS[.size]	{ <i>length</i>   <i>template-name</i> }
[ <i>macro-label</i> ]	DUMP	<i>filename</i>
[ <i>macro-label</i> ]	EJECT	[ <i>lines</i> ]
[ <i>macro-label</i> ]	END	
[ <i>macro-label</i> ]	ENDF [UNC]	
[ <i>macro-label</i> ]	ENDMAIN	
[ <i>macro-label</i> ]	ENDP [ROC]	
[ <i>macro-label</i> ]	ENDR	
[ <i>macro-label</i> ]	ENDWITH	
[ <i>macro-label</i> ]	ENTRY	$\left\{ \begin{array}{l} ( name_1, name_2, \dots ) : \left\{ \begin{array}{l} CODE \\ DATA \end{array} \right\} \\ name_i : \left[ \begin{array}{l} CODE \\ DATA \\ MAIN \end{array} \right], \dots \end{array} \right\}, \dots$
<i>name</i>	EQU	$\left\{ \begin{array}{l} arith-expr \\ reg \\ import-name \end{array} \right\}$
[ <i>macro-label</i> ]	ERRLOG	<i>str-expr</i>

[ <i>macro-label</i> ]	EXPORT	$\left\{ \begin{array}{l} (name_1, name_2, \dots) : \left\{ \begin{array}{l} \text{CODE} \\ \text{DATA} \end{array} \right\} \\ name_1 : \left[ \begin{array}{l} \text{CODE} \\ \text{DATA} \\ \text{MAIN} \end{array} \right], \dots \end{array} \right\}$
[ <i>macro-label</i> ]	FORWARD	$\left\{ \begin{array}{l} W[ \text{ORD} ] \\ L[ \text{ONG} ] \end{array} \right\}$
<i>name</i>	FREG	<i>fp-rlist</i>
[ <i>name</i> ]	FUNC	$\left[ \begin{array}{l} \text{ENTRY} \\ \text{EXPORT} \end{array} \right]$
[ <i>macro-label</i> ]	IMPORT	$\left\{ \begin{array}{l} (name_1, name_2, \dots) : \left\{ \begin{array}{l} \text{CODE} \\ \text{DATA} \\ \text{type} \end{array} \right\} \\ name_1 : \left[ \begin{array}{l} \text{CODE} \\ \text{DATA} \\ \text{type} \end{array} \right], \dots \end{array} \right\}, \dots$
[ <i>macro-label</i> ]	INCLUDE	<i>filename</i>
[ <i>macro-label</i> ]	LOAD	<i>filename</i>
[ <i>macro-label</i> ]	MACHINE	$\left\{ \begin{array}{l} \text{MC 68000} \\ \text{MC 68010} \\ \text{MC 68020} \\ \text{MC 68030} \end{array} \right\}$

[ name ]	MAIN	{ ENTRY } { EXPORT }
[ macro-label ]	MC68851	
[ macro-label ]	MC68881	[ /p-option ], ...
[ macro-label ]	OPT	{ ALL } { NONE } { NOCLR }
name	OPWORD	abs-expr
[ macro-label ]	ORG	[ expr ]
[ macro-label ]	PAGESIZE	[ lines ] [ , width ]
[ name ]	PROC	{ ENTRY } { EXPORT }
[ macro-label ]	PRINT	parameter, ...
[ name ]	RECORD	{ ENTRY } { EXPORT } [ , { INCR[ EMENT ] } { DECR[ EMENT ] } MAIN ]
name	RECORD	{ offset IMPORT { 'origin' } } [ , { INCR[ EMENT ] } { DECR[ EMENT ] } ]
name	REG	rlist
[ macro-label ]	SEG	[ str-expr ]

<i>name</i>	SET	$\left\{ \begin{array}{l} \textit{arith-expr} \\ \textit{reg} \\ \textit{import-name} \end{array} \right\}$
[ <i>macro-label</i> ]	SPACE	[ <i>lines</i> ]
[ <i>macro-label</i> ]	STRING	$\left\{ \begin{array}{l} \text{ASIS} \\ \text{PASCAL} \\ \text{C} \end{array} \right\}$
[ <i>macro-label</i> ]	TITLE	<i>str-expr</i>
[ <i>macro-label</i> ]	WITH	<i>name,...</i>

---

## Macro and SET variable directives

[ <i>macro-label</i> ]	ACTR	<i>arith-expr</i>
[ <i>macro-label</i> ]	ANOP	
[ <i>macro-label</i> ]	AERROR	<i>str-expr</i>
[ <i>macro-label</i> ]	{ EXITM   MEXIT }	
[ <i>macro-label</i> ]	$\left\{ \begin{array}{l} \text{LCLA} \\ \text{LCLC} \\ \text{GBLA} \\ \text{GBLC} \end{array} \right\}$	<i>set-var-name, ...</i>
[ <i>macro-label</i> ]	GOTO	[+   -]{ <i>macro-label</i>   <i>str-expr</i> }
[ <i>macro-label</i> ] <i>expr</i> }	IF	<i>bool-expr</i> GOTO [+   -]{ <i>macro-label</i>   <i>str-</i>



[macro-label] IF *bool-expr* THEN  
*statements*  
ELSEIF *bool-expr* THEN  
*statements*  
ELSE  
*statements*  
[macro-label] ENDI[F]

MACRO

[& name]      *name* . & name       $\left[ \begin{array}{c} \text{INT} \\ \text{STR} \\ \text{A} \\ \text{C} \end{array} \right]$        $\left[ \left\{ \text{=} \right\} \text{opnd-value} \right], \dots$

*machine instruction or directive statements*

[macro-label] { ENDM | MEND }

*set-var-name*      SETA      *arith-expr*

*set-var-name*      SETC      *str-expr*

[macro-label] WHILE *bool-expr* DO  
...  
assembler statements

...

[macro-label] ENDW[HILE]

[macro-label] WRITE      *item*,...

[macro-label] WRITELN      [*item*],...

---

## SET variable functions

&ABS (*arith-expr*)  
&CHR (*arith-expr*)  
&CONCAT (*str-expr*, ...)  
&DELSYMTBL (*sym-tbl*)  
&DEFAULT (*str-expr*<sub>1</sub>, *str-expr*<sub>2</sub>)  
&ENTERSYM (*sym-tbl*, *symbol*, *value*, *flags*)  
&EVAL (*str-expr*)  
&FINDSYM (*sym-tbl*, *symbol*)  
&GETENV (*str-expr*)  
{ &INTTOSTR | &I2S } (*arith-expr* [, *width* [, *hex*]])  
&ISINT (*str-expr*)  
&LEN (*str-expr*)  
&LEX (*str-expr*, *start*)  
&LIST (*str-expr*, *str-arr* [, *delimiter*])  
{ &LOWCASE | &LC } (*str-expr*)  
&MAX (*arith-expr*, ...)  
&MIN (*arith-expr*, ...)  
&NBR ({*symb-param* | &SYSLIST})  
&NEWSYMTBL  
&ORD (*expr*)  
&POS (*str-expr*<sub>1</sub>, *str-expr*<sub>2</sub>)  
&SCANEQ (*ch*, *str-expr*, *start*)  
&SCANNE (*ch*, *str-expr*, *start*)  
&SETTING (*str-expr* [, *arith-expr*])

<b>str-exp ::=</b>	<b>return value ::=</b>
ALIGN	0, 1
BLANKS	ON, OFF
BRANCH	SHORT, WORD, LONG
CASE	ON, OFF, OBJECT
CODEREFS	FORCEJT, NOFORCEJT, FORCEPC
DATAREFS	ABSOLUTE, RELATIVE
FORWARD	WORD, LONG
MACHINE	MC68000, MC68010, MC68020, MC68030
OPT	ALL, NONE, NOCLR
PRINT	ON, OFF, GEN, NOGEN, PAGE, NOPAGE, WARN, NOWARN, MCALL, NOMCALL, OBJ, NOOBJ, DATA, NCDATA, MDIR, NOMDIR, HDR, NOHDR, LITS, NOLITS, STAT, NOSTAT, SYM, NOSYM
STRING	PASCAL, ASIS, C

```

{ &STRTOINT | &S2I } (str-exp)
&SUBSTR(str-exp, start, length)
&TRIM(str-exp [ , trim-left ])
&TYPE (str-exp)

```

If *str-exp* is a macro variable name, then the return value ::=

```

UNDEFINED
PARAM [ STRUCTURED ] { INT | STR }
{ SETA | SETC } [ ARRAY['dim'] ]
MACRO [ { FUNCTION | SYSVAR } ]

```

If *str-exp* is a non-macro name, then value ::=

```

UNDEFINED
{ CODE | DATA } IMPORT
REG { AN | DN | ZAN | ZDN | CCR | SR | USP | MSP | SFC | DFC |
      CAAR | VBR | CACR | ISP | CRP | SRP | DRP | TC | PSR |
      PCSR | AC | CAL | SCC | VAL | BADn | BACn }
FPREG { FPn | FPCR | FPSR | FPIAR }
RLIST
FRLIST
FCRLIST
{ CODE | DATA } MODULE { EXPORT | ENTRY | IMPORT } [MAIN] ['('type')']
TEMPLATE [DATA IMPORT] ['('type')']
TEMPLATE FIELD ['('type ')']
DATA FIELD [ { EXPORT | ENTRY | IMPORT } ] ['('type')']
{ CODE | DATA } LABEL [ { EXPORT | ENTRY | IMPORT } ] [MAIN] ['('type')']
SET
EQU
OPWORD
{ &UPCASE | &UC } (str-exp)

```



## Appendix C Assembly Listing Format

WHEN THE MPW ASSEMBLER PRODUCES A LISTING, it normally follows the format shown in Figure C-1. This format is based on the assumption that you will handle the listing in one of the following ways:

- print it, using a monospace font
- edit it with an editor program that can scroll horizontally

The Assembler follows this format when all the listing-control default values are in effect; that is, when your source text includes only a `TITLE` directive and you specify no listing-control options in the Assembler command line. By using any other of the listing-control directives described in Chapter 4, or by using the listing-control Assembler options described in Appendix H, you can change the listing format. ■



■ **Figure C-1** Default assembly listing format

```

MC68020 Assembler - Ver v.rr <Title goes here>  dd-Mon-yy  Page xxx
Copyright Apple Computer, Inc. 1984 - 1988

Loc  F      Object Code      Addr  M      Source Statement
xxxxx  x      xxxxx xxxxx xxxxx  xxxxxx  x      |-- machine instruction ----->
                                     xxxxx xxxxx
                                     xxxxx
xxxxxx      xxxxx xxxxx xxxxx                x      |-- data statement ----->
                                     xxxxx xxxxx xxxxx
                                     ---
                                     ---
                                     ---
                                     xxxxx xxxxx xxxxx
                                     xxxxx xxxxx x...
```

Figure C-1 shows the layout of an assembly listing, including a typical header plus a few listing lines. The *x*'s in Figure C-1 indicate the number of characters generated in each section; other text indicates the type of information listed in that section. The six header lines (three of which are always blank) appear at the top of every page.

The first header line contains only the form feed character (ASCII \$0C) that ejects each page.

The second header line contains the version and revision number of the Assembler, the title (if you wrote a `TITLE` directive in your source text), the date, and the page number. The Assembler formats this information according to these rules:

- The date and page number are right-justified on the page in conformance with the current `PAGESIZE` directive width setting.
- The title is truncated if it is too long for the available space.

The third header line always contains Apple's copyright notice.

The fourth and sixth header lines are always blank.

The fifth header line always contains the column headings for the listing. The *x*'s under the first five headings in Figure C-1 tell you the maximum number of characters that may appear in these columns. The six headings identify parts of each listing line, as described in the following paragraphs.

The Loc column identifies the location of the generated object code in the code or data module. This field is truncated to five hexadecimal digits or the number of digits you specified with an `-addrsize` option when invoking the Assembler (see "Assembler Options" in Appendix G). It is blank if the corresponding source text does not generate any object code. There is no guarantee that this location is correct if an Assembler error has occurred.

The F column contains generic and privileged instruction flags. If the Assembler converts a machine instruction from generic form, as described in Appendix A, the letter *G* appears in this column. If a machine instruction is privileged, the letter *P* appears. (Privileged instructions are described in the Motorola *M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual*.) In all other cases this column is blank.

The Object Code column contains the generated object code for the source line. This column is formatted one of three ways, depending on whether it contains a machine instruction, DC-generated data, or DCB-generated data:

- With machine instructions, all generated code is shown. The first line contains up to three words of four hexadecimal digits each. Each subsequent line, if any, contains up to two words, indented by one word.
- With data generated by a DC directive, every line contains up to three words. If you specified `PRINT NODATA`, then only the first line (or one line for each continuation) will appear. If you specified `PRINT DATA`, then up to 18 lines of data will appear. Any data remaining after 18 lines will be indicated by three periods, as illustrated at the bottom of Figure C-1.
- Data generated by a DCB directive appears in the same form as data generated by a DC directive; but instead of appearing only as words, it appears as bytes, words, or long words, depending on the directive's modifier, with a space between each text item. Such data listings are also limited to 18 lines.

The Object Code column also lists the values of defined symbols and SET variables. Register equates give the register name if the equate is to an A or D register; control registers are not shown. REG and FREG directive register-mask equates show a value consisting of two register masks, one in each direction, depending on the form of the MOVEM or FMOVEM instruction using the mask. EXPORT and ENTRY directives inside modules give the Loc-column locations corresponding to the specified labels.



The Addr column contains addresses accessed by machine instructions that have nonimported PC-relative address operands. It tells you the Loc-column value referenced by the instruction. As with the Loc column, the Addr-column value is truncated to five digits or to the number of digits specified by an `-addrsiz` Assembler option. All other addressing modes that access locations express offsets and so may be seen in the object code itself. If the instruction does not have a PC-relative address operand, or if its operand is a PC-relative reference to an imported identifier, this column is blank.

The M column lists the dynamic macro-nesting level, reduced modulo 10. If the listing line was not generated from a macro, the M column is blank.

The Source Statement column contains your source text line.

- ◆ *Note:* If you specified `PRINT NOOBJ`, either by a directive or by an Assembler option, then only the Loc and Source Statement columns will be displayed. This produces a more compressed listing with less information.

When the Assembler creates a listing file, it defines the creator as 'MPS ' and the type as 'TEXT'. This lets you use the Macintosh Programmer's Workshop facilities to edit the listing. It also creates the following MPW Editor resource information to facilitate editing with the MPW Editor and printing with the MPW `Print` command:

- The listing's tabs are set to the same locations as the original source text tabs, or every four columns if the source text has no tabs. The Assembler places a tab in the listing file just before each entry in the source statement column. This gives the source statements in the listing the same format they had when being edited.
- The listing's font and font size are set according to the `-font` Assembler option. The standard default font is 7-point Monaco. However, a LaserWriter printer will adopt Courier as a default font when printing the listing.



## Appendix D Other Assemblers

BEFORE THE RELEASE OF MPW, THE PRIMARY DEVELOPMENT ENVIRONMENTS for the Macintosh were the Lisa Workshop (which included the TLA Assembler) and the Macintosh 68000 Development System (MDS). This appendix compares MPW with these other two assemblers and with the Motorola assembler. ■

### *Contents*

Syntax comparison	213
Writing identifiers	213
Writing numbers	214
Writing strings	214
Defining modules	215
Communicating between modules	215
Writing expressions	215
Location-counter reference	216
Addressing features	217
Writing macros	217



---

## Syntax comparison

This section compares the syntax accepted by the MPW Assembler with that accepted by the following other assemblers:

- The Motorola assembler (Mot)
- Apple MDS
- Apple TLA

---

## Writing identifiers

Table D-1 shows which characters may be used to compose identifiers in the four assembly languages, and how long those identifiers may be.

■ **Table D-1** Identifier syntax rules

Rule	MPW	Mot	MDS	TLA
Lowercase letters (a..z)	Yes*	Yes*	Yes*	Yes*
Uppercase letters (A..Z)	Yes*	Yes*	Yes*	Yes*
Digits (0..9)	Yes	Yes	Yes	Yes
Underscores ( _ )	Yes*	Yes	Yes*	Yes*
Periods ( . )	No	Yes	Yes*	Yes
At symbols ( @ )	Yes†	No	Yes†	Yes†
Dollar signs ( \$ )	Yes	No	Yes	No
Number signs ( # )	Yes	No	No	No
Percent signs ( % )	Yes*,‡	No	No	Yes*,‡
Maximum length	63	8	∞	8

\* Identifiers may start with these characters.

† The at symbol is used to begin @-labels. It may be embedded inside identifiers with the MPW Assembler.

‡ Leading percent signs should be avoided. They are reserved by Apple for special software such as the Pascal runtime system. In the MPW Assembler, identifiers beginning with % may not use 0 or 1 for a second character because % is used to indicate binary numbers.

---

## Writing numbers

Table D-2 shows how numbers are written in the four assembly languages. In this table, *d...* represents any sequence of integers.

■ **Table D-2** Number syntax

Numerical base	MPW	Mot	MDS	TLA
Decimal	<i>d...</i>	<i>d...</i>	<i>d...</i>	<i>d...</i>
Hexadecimal	<i>\$d...</i>	<i>\$d...</i>	<i>\$d...</i>	<i>\$d...</i> or <i>d...H</i>
Binary	<i>%d...</i>	<i>%d...</i>	<i>%d...</i>	<i>d...B</i>
Octal	<i>@d...</i>	<i>^d...</i>	<i>d...O</i>	
Floating-point	<i>"..."</i>	<i>±d.d... or d...</i>		

- Written as decimal or hexadecimal strings enclosed in quotation marks. Floating-point string formats may be any of those shown in the Apple Numerics Manual.

---

## Writing strings

Table D-3 gives the rules and capabilities for writing strings in the four assembly languages. In this table, the ellipsis (...) indicates a sequence of characters.

■ **Table D-3** String syntax

Syntax rule	MPW	Mot	MDS	TLA
Form	'...'	'...'	'...'	'...!' or "..."
Apostrophe representation	"	"	"	"...!"
Ampersand representation	&&*	&	&	&
Generates Pascal strings	Yes	No	Yes	No
Generates C strings	Yes	No	Yes	No

- In macros only.

---

## Defining modules

Table D-4 compares the way that the four assembly languages define code and data modules in source text.

■ **Table D-4** Module definition

Module	MPW	Mot	MDS	TLA
Code	PROC . . . ENDP*	Control section	One module†	PROC‡
Data	RECORD . . . ENDR*	Control section	DS directives§	None

\* Modules may be local to a file.

† MODULE directive permits multiple code modules in MDS 2.0.

‡ Modules are always exported.

§ Only DS directives generate A5-relative data.

---

## Communicating between modules

Table D-5 compares the directives that allow each assembly language to transfer references to the source text between modules.

■ **Table D-5** Communication directives

MPW	Mot	MDS	TLA
EXPORT	XDEF	XDEF	DEF
IMPORT	XREF	XREF	REF, REFA5, REF32
ENTRY			

---

## Writing expressions

Table D-6 lists the operators you can use when writing expressions in each of the four assembly languages. Some MPW operators may be written in more than one way. All alternatives are shown.

■ **Table D-6** Allowable operators

Operation		MPW*		Motor*	MDS*	TLA†
Addition	+			+	+	+
Subtraction	-			-	-	-
Multiplication	*			*	*	*
Division	/	DIV	+	/	/	/
Modulus reduction	//	MOD			\	
Logical or	++	OR		!	!	
Exclusive-or	--	XOR			^	
Logical and	**	AND		&	&	
Equal to	=					=
Not equal to	<>	≠			<>	
Less than	<					
Greater than	>					
Less than or equal to	<=	≤				
Greater than or equal to	>=	≥				
Shift left	<<			<<	<<	
Shift right	>>			>>	>>	
One's complement	-					-
Negation	-			-	-	-
Logical not	¬	NOT				

\* Operators have precedence; parentheses are allowed.

† No operator precedence; no parentheses allowed. Angle brackets (greater than and less than symbols) are used in place of parentheses.

### Location-counter reference

The meaning of the location-counter symbol (\*) varies between the four assembly languages when used in the DC directive (WORD or LONG in TLA). For example, in the statement

```
label DC.W 1,2,*-X,3,4
```

the value of \* is defined as the value of the label—the location of the first word—in the MPW, Motorola, and TLA assemblers. In MDS, the value of \* is the location of the word representing the current operand. Therefore in MDS, \* represents the location of the third word.



---

## Addressing features

The four assembly languages include different features for writing addresses, as shown in Table D-7.

■ **Table D-7** Addressing features

Feature	MPW	Mot	MDS	TLA
MC68020 addressing	Yes	Yes	No	No
Bases	A5 default	A5 for data	A5 for DS	No
Qualified identifiers	Yes	No	No	No
Data structures	Templates	OFFSET, EQU	EQU	EQU

---

## Writing macros

The MPW Assembler Macro Processor does not accept macro definitions written in any of the other three languages. In all four assemblers, however, macro calls have the same basic form.

The MPW Assembler's Macro Processor supports all the features of the other assemblers with one exception: macros that generate only part of a statement (such as only an operand) are not supported. Such macros can be written only in the MDS Assembler. The MPW Assembler also supports a number of features not found in any of the other assemblers, such as keyword macros and SET variables.

Hence you can always rewrite Motorola, TLA, and MDS macro definitions into MPW form, except for MDS macros that generate partial source lines. In most cases, you can leave macro call directives as they were originally written.



## Appendix E The Macintosh Character Set

THE MACINTOSH CHARACTER SET IS INCLUDED HERE for  
your convenience. ■



		First digit															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Second digit	0	NLE	DLE	SPACE	0	@	P	`	p	Ã	ê	†	•	¿	-		
	1	SCH	DC1	!	1	A	Q	a	q	À	ë		±	ı	—		
	2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	£	¬	"		
	3	ETX	DC3	#	3	C	S	c	s	É	ì	£		÷	"		
	4	EOT	DCA	\$	4	D	T	d	t	Ñ	î	§	¥	f	'		
	5	ENQ	NAK	%	5	E	U	e	u	Ö	ï		m	'	'		
	6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	d	D			
	7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	Â	•	‡		
	8	BS	CAN	(	8	H	X	h	x	à	ò	®	'	•	ÿ		
	9	HT	EM	)	9	I	Y	i	y	â	ô	©	p	...			
	A	LF	SUB	•	:	J	Z	j	z	ä	ö	™	Ú	—			
	B	VT	ESC	+	;	K	[	k	{	â	ö	'	'	À			
	C	FF	FS	,	<	L	\	l		á	ú	"	°	Ã			
	D	CR	GS	-	=	M	]	m	}	ç	ù		W	Õ			
	E	SO	RS	.	>	N	^	n	-	é	û	Æ	æ	Œ			
	F	SI	LS	/	?	O	_	o	DEL	è	ü	Ø	ø	œ			

— Stands for a nonbreaking space, the same width as a digit.

☐ The dark-shaded characters cannot normally be generated from the Macintosh keyboard or keypad.



## Appendix F Instruction Sets

THIS APPENDIX DEFINES THE INSTRUCTION SETS accepted by the MPW Assembler. They are equivalent to the MC68000, MC68010, MC68020, MC68030, MC68881/MC68882, and MC68851 instruction sets described in more detail in the Motorola *MC68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual*, the Motorola *MC68881 Floating-Point Coprocessor User's Manual*, and the Motorola *MC68851 Paged Memory Management Unit User's Manual*. Refer to those manuals for full descriptions of these instructions.

- ◆ *Note:* Some mnemonics have been changed to eliminate ambiguities or to conform to the Motorola assembler forms. If in doubt, check your mnemonics with those given later in this appendix (in Tables F-7, F-8, and F-9).

The Macintosh instruction sets contain certain machine instructions that encode into more than one bit configuration, depending on the instruction's operands. Each instruction consists of an opcode word and zero or more extension words. The opcode word contains some basic constant information about the instruction, but other fields must be set to indicate the kinds of operands (effective addresses) and the size of the instruction. ■

### **Contents**

Instruction evaluation	225
Listing conventions	225
Opcode	226
Operands	226
Opcode word	227
Cp type	228
Group	228
Flags	228
Range	229
Equivalent	229
Condition codes	229
Instruction set listings	233





---

## Instruction evaluation

The Assembler determines the encoding for an instruction by looking at the group corresponding to the mnemonic. Starting with the first encoding line in the group, the Assembler checks the machine type, the size, the source operand mode, the destination operand mode, and (where applicable) the immediate data range. If all the information matches, the Assembler generates the code for the instruction, through its encoding group number. If any one of the items doesn't match, the next encoding (if any) in the group is checked. This process continues until an encoding is found or the end of the group is reached.

If the Assembler reaches the end of the group before finding a valid encoding (including coprocessor opcodes), it indicates an invalid instruction. It then tries to interpret the source text line as a macro. Finally, it tries to interpret it as an `OPWORD` directive.

---

## Listing conventions

Tables F-2 through F-9 list the instructions and condition codes accepted by the MPW Assembler. Table F-7, covering the processor instructions for the MC68000, 68010, and 68020, is divided into seven columns with the following headings:

- **Opcode:** The mnemonic you write in your source text.
- **Operands:** The operands (if any) required by the opcode.
- **Opcode word:** The binary encoding of the first word or extension word of the instruction.
- **Group:** The encoding group number assigned to the instruction.
- **Flags:** Letters indicating specific characteristics of the instruction.
- **Range:** A code number identifying the instruction's data range.
- **Equivalent:** The actual code equivalent for generic instructions.

Tables F-8 and F-9, covering the coprocessor instructions, have a slightly different set of column headings:

- **Opcode:** The mnemonic you write in your source text.
- **Operands:** The operands (if any) required by the opcode.
- **Opcode word:** The binary encoding of the first word of the instruction.
- **Cp type:** The coprocessor instruction type.
- **Group:** The encoding group number assigned to the instruction.
- **Flags:** Letters indicating specific characteristics of the instruction.
- **Equivalent:** (Table F-8 only.) The actual code equivalent for generic instructions.

The columns are described in more detail in the following sections.

---

## Opcode

This column contains the legal instruction mnemonics recognized by the Assembler. For further information about the instructions they represent, see the appropriate Motorola manual listed at the beginning of this appendix.

---

## Operands

This column may contain register names (such as  $Dn$  or  $A_n$ ) or nonterminal symbols (such as *ean* or *Rc*). The nonterminal symbols stand for addressing modes. Table F-1 shows all possible operand forms. The number *1* in the table indicates that the corresponding addressing mode is legal. The number *0* means it is illegal. Addressing modes are further described in the Motorola manuals and in Chapter 3 of this manual.

■ Table F-1 Instruction operands

Operand	ea0	ea1	ea2	ea3	ea4	ea5	ea6	ea7	ea8	ea9	ea10	ea11	ea12	Rc
Special	0	0	0	0	0	0	0	0	0	0	0	0	0	0
imm	1	1	0	1	0	0	1	0	0	1	1	0	0	0
imm	1	1	0	0	0	0	0	0	0	0	0	0	0	0
(An)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
(An) +	1	1	1	1	0	1	1	0	1	0	0	0	1	0
-(An)	1	1	1	1	1	1	1	0	0	0	0	0	1	0
d(An)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
d(An, Xn)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
(bd, An, Xn)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
([bd, An, Xn], od)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
([bd, An], Xn, od)	1	1	1	1	1	1	1	1	1	1	1	1	1	0
(ae).W   (ae).L	1	1	1	1	1	1	1	1	1	1	1	1	1	0
#data	1	0	0	0	0	0	1	0	0	0	0	0	1	0
label	1	0	0	0	0	1	1	1	1	0	1	0	1	0
d(PC)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
d(PC, Xn)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
(bd, PC, Xn)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
([bd, PC, Xn], od)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
([bd, PC], Xn, od)	1	0	0	0	0	1	1	1	1	0	1	0	1	0
RList	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CCR	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SR	0	0	0	0	0	0	0	0	0	0	0	0	0	0
USP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Ctl Regs*	0	0	0	0	0	0	0	0	0	0	0	0	0	1

\* Rc ::= SFC | DFC | CACR | VBR | CAAR | MSP | ISP

### Opcode word

This column contains the binary encoding of the fixed information placed in the first word of the instruction corresponding to the specified mnemonic.

---

## Cp type

This column lists the instruction type for coprocessor instructions. The meanings of the type codes in this column are shown here:

Type	Meaning
Gen1	General coprocessor instruction; register operand or operands, or no operand
Gen2	General coprocessor instruction; memory to register or registers
Gen3	General coprocessor instruction; register or registers to memory
Bcc	Branch on coprocessor condition
DBcc	Decrement and branch on coprocessor condition
Rest	Coprocessor restore instruction (privileged)
Save	Coprocessor save instruction (privileged)
Scc	Set on coprocessor condition
Tcc1	Trap on coprocessor condition; predicate supplied by processor
Tcc2	Trap on coprocessor condition; operand predicate

---

## Group

This column lists the Assembler's encoding group for that mnemonic. The set of all MPW Assembler instructions may be viewed as a collection of subsets, with each subset corresponding to a specific encoding. There are 49 distinct encoding groups, numbered 0 to 48.

---

## Flags

This column indicates various attributes about the instruction. The meanings of the flag symbols are given here:

Flag	Meaning
1	MC68010 instruction
2	MC68020 instruction
P	Privileged instruction
G	Generic instruction
B	Byte data size accepted
W	Word data size accepted
L	Long-word data size accepted
S	Single-precision data size accepted
D	Double-precision data size accepted
X	Extended data size accepted
K	Packed BCD data size accepted

---

## Range

This column specifies the legal range for absolute data, if the range is not otherwise expressed. The codes that specify the ranges are described here:

Number	Meaning
0	Value = 0
3	Value ::= 1..8; 0 indicates a value of eight
4	4-bit unsigned value
8	8-bit unsigned value
-8	8-bit signed value
16	16-bit unsigned value
-16	16-bit signed value
32	32-bit unsigned value

---

## Equivalent

This column indicates what actual instruction the mnemonic represents in cases where the instruction form is generic. Generic instructions are described in Appendix A.

---

## Condition codes

Tables F-2 through F-6 list only the condition codes that the MPW Assembler accepts. Tables F-7, F-8, and F-9 list *all combinations of instructions and condition codes* that the MPW Assembler accepts.

■ Table F-2 MC68xxx condition codes

Mnemonic	Condition	Encoding	Test
T	True	0000	1
HI	High	0010	$\overline{C} \cdot \overline{Z}$
LS	Low or same	0011	$C + Z$
CC, CS	Carry clear	0100	$\overline{C}$
CS, LO	Carry set	0101	C
ME, NZ	Not equal	0110	$\overline{Z}$
EQ, Z	Equal	0111	Z
VC	Overflow clear	1000	$\overline{V}$
VS	Overflow set	1001	V
PL	Plus	1010	$\overline{N}$
MI	Minus	1011	N
GE	Greater than or equal to	1100	$\overline{N} \cdot \overline{V} + N \cdot V$
LT	Less than	1101	$\overline{N} \cdot \overline{V} + N \cdot V$
GT	Greater than	1110	$\overline{N} \cdot \overline{V} \cdot \overline{Z} + N \cdot V \cdot Z$
LE	Less than or equal to	1111	$\overline{Z} + \overline{N} \cdot V + N \cdot V$

■ Table F-3 MC68881 IEEE nonaware tests

Mnemonic	Definition	Equation	Predicate
EQ	Equal	Z	000001
NE	Not equal	$\bar{Z}$	001110
GT	Greater than	$\overline{NAN+Z+N}$	010010
NGT	Not greater than	NAN+Z+N	011101
GE	Greater than or equal to	$Z + \overline{NAN+N}$	010011
NGE	Not (greater than or equal to)	NAN+ (N • Z)	011100
LT	Less than	$N \cdot \overline{NAN+Z}$	010100
NLT	Not less than	NAN+Z+N	011011
LE	Less than or equal to	$Z + (N + \overline{NAN})$	010101
NLE	Not (less than or equal to)	NAN+ $\overline{(N+Z)}$	011010
GL	Greater than or less than	$\overline{NAN+Z}$	010110
NGL	Not (greater than or less than)	NAN+Z	011001
GLE	Greater than, less than, or equal to	$\overline{NAN}$	010111
NGLE	Not (greater than, less than, or equal to)	NAN	011000

■ Table F-4 MC68881 IEEE aware tests

Mnemonic	Definition	Equation	Predicate
EQ	Equal	Z	000001
NE	Not equal	$\bar{Z}$	001110
OGT	Ordered greater than	$\overline{NAN+Z+N}$	000010
ULE	Unordered less than or equal to	NAN+Z+N	001101
OGE	Ordered greater than or equal to	$Z + \overline{(NAN+N)}$	000011
ULT	Unordered less than	NAN+ (N • Z)	001100
OLT	Ordered less than	$N \cdot \overline{(NAN+Z)}$	000100
UGE	Unordered or greater than or equal to	NAN+Z+N	001011
OLE	Ordered less than or equal to	$Z + (N + \overline{NAN})$	000101

(continued)

■ Table F-4 (continued) MC68881 IEEE aware tests

Mnemonic	Definition	Equation	Predicate
UGT	Unordered or greater than	$\overline{\text{NAN}} + (\text{N} + \text{Z})$	001010
OGL	Ordered greater than or less than	$\overline{\text{NAN} + \text{Z}}$	000110
UEQ	Unordered or equal to	$\text{NAN} + \text{Z}$	001001
OR	Ordered	$\overline{\text{NAN}}$	000111
UN	Unordered	$\text{NAN}$	001000

■ Table F-5 MC68881 miscellaneous tests

Mnemonic	Definition	Equation	Predicate
F	False	False	000000
T	True	True	001111
SF	Signaling false	False	010000
ST	Signaling true	True	011111
SEQ	Signaling equal	Z	010001
SNE	Signaling not equal	Z	011110

■ Table F-6 MC68851 PMMU condition codes

Mnemonic	Condition	Encoding
BS	B set	000000
BC	B clear	000001
LS	L set	000010
LC	L clear	000011
SS	S set	000100
SC	S clear	000101
AS	A set	000110
AC	A clear	000111
WS	W set	001000
WC	W clear	001001
IS	I set	001010
IC	I clear	001011
GS	G set	001100
GC	G clear	001101
CS	C set	001110
CC	C clear	001111



## Instruction set listings

Tables F-7, F-8, and F-9 are edited listings of the actual data files used to produce the opcode table used by the Assembler. These tables show all the opcodes sorted alphabetically. Different encodings for the same mnemonic are grouped, with a blank line separating each group. The encodings within each group are ordered so that generic forms or optimizations occur before more general forms.

In Tables F-8 and F-9, the following metasymbols are used to denote groups of coprocessor registers:

<i>FPn</i>	::=	FP0..FP7
<i>FRList</i>	::=	Floating-point register list
<i>FCRList</i>	::=	Floating-point control register list
<i>BADn</i>	::=	BAD0..BAD7
<i>BACn</i>	::=	BAC0..BAC7
<i>xRP</i>	::=	CRP   SRP   DRP
<i>SCCCAL</i>	::=	SCC   CAL

■ Table F-7 MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
ABCD	<i>Dn, Dn</i>	1100 000 100 000 000	6	B		
ABCD	<i>-(An), -(An)</i>	1100 000 100 001 000	6	B		
ADD	<i>#data, &lt;ea1&gt;</i>	0101 000 000 000 000	24	BWLG	3	ADDQ (Opt)
ADD	<i>#data, &lt;ea3&gt;</i>	0000 011 000 000 000	25	BWLG		ADDI
ADD	<i>&lt;ea0&gt;, Dn</i>	1101 000 000 000 000	22	BWL		
ADD	<i>Dn, &lt;ea2&gt;</i>	1101 000 100 000 000	23	BWL		
ADD	<i>&lt;ea0&gt;, An</i>	1101 000 011 000 000	27	WLG		ADDA
ADDA	<i>#data, &lt;ea1&gt;</i>	0101 000 000 000 000	24	WLG	3	ADDQ (Opt)
ADDA	<i>&lt;ea0&gt;, An</i>	1101 000 011 000 000	27	WL		
ADDI	<i>#data, &lt;ea3&gt;</i>	0000 011 000 000 000	25	BWL		
ADDQ	<i>#data, &lt;ea1&gt;</i>	0101 000 000 000 000	24	BWL	3	
ADDX	<i>Dn, Dn</i>	1101 000 100 000 000	8	BWL		
ADDX	<i>-(An), -(An)</i>	1101 000 100 001 000	8	BWL		
AND	<i>#data, &lt;ea3&gt;</i>	0000 001 000 000 000	25	BWLG		ANDI
AND	<i>Dn, &lt;ea2&gt;</i>	1100 000 100 000 000	23	BWL		
AND	<i>&lt;ea6&gt;, Dn</i>	1100 000 000 000 000	22	BWL		

(continued)

■ Table F-7 (continued)

MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
ANDI	#data, CCR	0000 001 000 111 100	1	B	8	
ANDI	#data, SR	0000 001 001 111 100	1	PW	16	
ANDI	#data, <ea3>	0000 001 000 000 000	25	BWL		
ASL	#data, Dn	1110 000 100 000 000	10	BWL	3	
ASL	Dn, Dn	1110 000 100 100 000	9	BWL		
ASL	<ea2>	1110 000 111 000 000	15	W		
ASR	#data, Dn	1110 000 000 000 000	10	BW	3	
ASR	Dn, Dn	1110 000 000 100 000	9	BWL		
ASR	<ea2>	1110 000 011 000 000	15	W		
BCC	label	0110 0100 00000000	14	BWL		
BCHG	#data, Dn	0000 100 001 000 000	19	L		
BCHG	#data, <ea2>	0000 100 001 000 000	19	B		
BCHG	Dn, Dn	0000 000 101 000 000	20	L		
BCHG	Dn, <ea2>	0000 000 101 000 000	20	B		
BCLR	#data, Dn	0000 100 010 000 000	19	L		
BCLR	#data, <ea2>	0000 100 010 000 000	19	B		
BCLR	Dn, Dn	0000 000 110 000 000	20	L		
BCLR	Dn, <ea2>	0000 000 110 000 000	20	B		
BCS	label	0110 0101 00000000	14	BWL		
BEQ	label	0110 0111 00000000	14	BWL		
BFCHG	special	1110 101 011 000 000	31	2		
BFCLR	special	1110 110 011 000 000	31	2		
BFEXTS	special	1110 101 111 000 000	32	2		
BFEXTU	special	1110 100 111 000 000	32	2		
BFFFO	special	1110 110 111 000 000	32	2		
BFINS	special	1110 111 111 000 000	33	2		
BFSET	special	1110 111 011 000 000	31	2		
BFTST	special	1110 100 011 000 000	31	2		
BGE	label	0110 1100 00000000	14	BWL		
BGT	label	0110 1110 00000000	14	BWL		
BHI	label	0110 0010 00000000	14	BWL		
BHS	label	0110 0100 00000000	14	BWLG		BCC
BKPT	#data	0100 100 001 001 000	2	2	4	

■ Table F-7 (continued)

MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
BLE	<i>label</i>	0110 1111 00000000	14	BWL		
BLO	<i>label</i>	0110 0101 00000000	14	BWLG		BCS
BLS	<i>label</i>	0110 0011 00000000	14	BWL		
BLT	<i>label</i>	0110 1101 00000000	14	BWL		
BMI	<i>label</i>	0110 1011 00000000	14	BWL		
BNE	<i>label</i>	0110 0110 00000000	14	BWL		
BNZ	<i>label</i>	0110 0110 00000000	14	BWLG		BNE
BPL	<i>label</i>	0110 1010 00000000	14	BWL		
BRA	<i>label</i>	0110 0000 00000000	14	BWL		
BSET	<i>*data, Dn</i>	0000 100 011 000 000	19	L		
BSET	<i>*data, &lt;ea2&gt;</i>	0000 100 011 000 000	19	B		
BSET	<i>Dn, Dn</i>	0000 000 111 000 000	20	L		
BSET	<i>Dn, &lt;ea2&gt;</i>	0000 000 111 000 000	20	B		
BSR	<i>label</i>	0110 0001 00000000	14	BWL		
BT	<i>label</i>	0110 0000 00000000	14	BWLG		BRA
BTST	<i>*data, Dn</i>	0000 100 000 000 000	19	L		
BTST	<i>*data, &lt;ea5&gt;</i>	0000 100 000 000 000	19	B		
BTST	<i>Dn, Dn</i>	0000 000 100 000 000	20	L		
BTST	<i>Dn, &lt;ea12&gt;</i>	0000 000 100 000 000	20	B		
BVC	<i>label</i>	0110 1000 00000000	14	BWL		
BVS	<i>label</i>	0110 1001 00000000	14	BWL		
BZ	<i>label</i>	0110 0111 00000000	14	BWL		BEQ
CALLM	<i>*data, &lt;ea7&gt;</i>	0000 011 011 000 000	47	2		
CAS	<i>special</i>	0000 100 011 000 000	34	2BWL		
CAS2	<i>special</i>	0000 100 011 111 100	35	2BWL		
CHK	<i>&lt;ea6&gt;, Dn</i>	0100 000 110 000 000	21	W		
CHK	<i>&lt;ea6&gt;, Dn</i>	0100 000 100 000 000	21	2L		
CHK2	<i>&lt;ea7&gt;, Dn</i>	0000 000 011 000 000	36	2BWL		
CHK2	<i>&lt;ea7&gt;, An</i>	0000 000 011 000 000	36	2BWL		
CLR	<i>Dn</i>	0111 000 0 00000000	11	LG		MOVEQ (Opt)
CLR	<i>&lt;ea3&gt;</i>	0100 001 000 000 000	17	BWL		

(continued)

■ Table F-7 (continued)

MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalents
CMP	#data, <ea3>	0000 110 000 000 000	25	BWLG		CMPI
CMP	<ea0>, Dn	1011 000 000 000 000	22	BWL		
CMP	<ea0>, An	1011 000 011 000 000	27	WLG		CMPA
CMP	(An)+, (An)+	1011 000 100 001 000	6	WG		CMPM
CMP	(An)+, (An)+	1011 000 101 001 000	6	BG		CMPM
CMP	(An)+, (An)+	1011 000 110 001 000	6	LG		CMPM
CMPA	<ea0>, An	1011 000 011 000 000	27	WL		
CMPI	#data, <ea3>	0000 110 000 000 000	25	BWL		
CMPM	(An)+, (An)+	1011 000 100 001 000	6	W		
CMPM	(An)+, (An)+	1011 000 101 001 000	6	B		
CMPM	(An)+, (An)+	1011 000 110 001 000	6	L		
CMP2	<ea7>, Dn	0000 000 011 000 000	37	2BWL		
CMP2	<ea7>, An	0000 000 011 000 000	37	2BWL		
DBCC	Dn, label	0101 0100 11 001 000	13	W		
DBCS	Dn, label	0101 0101 11 001 000	13	W		
DBEQ	Dn, label	0101 0111 11 001 000	13	W		
DBF	Dn, label	0101 0001 11 001 000	13	W		
DBGE	Dn, label	0101 1100 11 001 000	13	W		
DBGT	Dn, label	0101 1110 11 001 000	13	W		
DBHI	Dn, label	0101 0010 11 001 000	13	W		
DBHS	Dn, label	0101 0100 11 001 000	13	WG		DBCC
DBLE	Dn, label	0101 1111 11 001 000	13	W		
DBLO	Dn, label	0101 0101 11 001 000	13	WG		DBCS
DBLS	Dn, label	0101 0011 11 001 000	13	W		
DBLT	Dn, label	0101 1101 11 001 000	13	W		
DBMI	Dn, label	0101 1011 11 001 000	13	W		
DBNE	Dn, label	0101 0110 11 001 000	13	W		
DBNZ	Dn, label	0101 0110 11 001 000	13	WG		DBNE
DBPL	Dn, label	0101 1010 11 001 000	13	W		
DBRA	Dn, label	0101 0001 11 001 000	13	WG		DBF
DBT	Dn, label	0101 0000 11 001 000	13	W		
DBVC	Dn, label	0101 1000 11 001 000	13	W		

■ Table F-7 (continued)

MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
DBVS	<i>Dn, label</i>	0101 1001 11 001 000	13	W		
DBZ	<i>Dn, label</i>	0101 0111 11 001 000	13	W		DBEQ
DIVS	<i>special</i>	0100 110 001 000 000	38	2L		
DIVS	<i>&lt;ea6&gt;, Dn</i>	1000 000 111 000 000	21	W		
DIVU	<i>special</i>	0100 110 001 000 000	39	2L		
DIVU	<i>&lt;ea6&gt;, Dn</i>	1000 000 011 000 000	21	W		
EOR	<i>#data, &lt;ea3&gt;</i>	0000 101 000 000 000	25	BWLG		EORI
EOR	<i>Dn, &lt;ea3&gt;</i>	1011 000 100 000 000	23	BWL		
EORI	<i>#data, &lt;ea3&gt;</i>	0000 101 000 000 000	25	BWL		
EORI	<i>#data, CCR</i>	0000 101 000 111 100	1	B	8	
EORI	<i>#data, SR</i>	0000 101 001 111 100	1	PW	16	
EXG	<i>An, Dn</i>	1100 000 110 001 000	6	L		
EXG	<i>Dn, Dn</i>	1100 000 101 000 000	7	L		
EXG	<i>An, An</i>	1100 000 101 001 000	7	L		
EXG	<i>Dn, An</i>	1100 000 110 001 000	7	L		
EXT	<i>Dn</i>	0100 100 010 000 000	2	W		
EXT	<i>Dn</i>	0100 100 011 000 000	2	L		
EXTB	<i>Dn</i>	0100 100 010 000 000	2	WG		EXT.W
EXTB	<i>Dn</i>	0100 100 111 000 000	2	2L		
EXTW	<i>Dn</i>	0100 100 011 000 000	2	LG		EXT.L
ILLEGAL		0100 101 011 111 100	0			
JMP	<i>&lt;ea7&gt;</i>	0100 111 011 000 000	15	W		
JSR	<i>&lt;ea7&gt;</i>	0100 111 010 000 000	15	W		
LEA	<i>&lt;ea7&gt;, An</i>	0100 000 111 000 000	46	L		
LINK	<i>An, #data</i>	0100 111 001 010 000	4	W	-16	
LINK	<i>An, #data</i>	0100 100 000 001 000	4	L		
LSL	<i>#data, Dn</i>	1110 000 100 001 000	10	BWL	3	
LSL	<i>Dn, Dn</i>	1110 000 100 101 000	9	BWL		
LSL	<i>&lt;ea2&gt;</i>	1110 001 111 000 000	15	W		
LSR	<i>#data, Dn</i>	1110 000 000 001 000	10	BWL	3	
LSR	<i>Dn, Dn</i>	1110 000 000 101 000	9	BWL		
LSR	<i>&lt;ea2&gt;</i>	1110 001 011 000 000	15	W		

(continued)

■ Table F-7 (continued)

MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
MOVE	#data, Dn	0111 000 0 00000000	11	LG	-8	MOVEQ (Opt)
MOVE	#data, <ea3>	0100 001 000 000 000	18	BWLC	0	CLR (Opt)
MOVE	#data, An	1001 000 111 000 000	28	WLG	0	SUBA.L (Opt)
MOVE	<ea0>, <ea3>	0000 000 000 000 000	26	BWL		
MOVE	<ea0>, An	0000 000 000 000 000	26	WLG		MOVEA
MOVE	An, USP	0100 111 001 100 000	2	PL		
MOVE	USP, An	0100 111 001 101 000	3	PL		
MOVE	<ea6>, CCR	0100 011 011 000 000	15	W		
MOVE	CCR, <ea3>	0100 001 011 000 000	16	1 W		
MOVE	<ea6>, SR	0100 010 011 000 000	15	PW		
MOVE	SR, <ea3>	0100 000 011 000 000	16	PW		
MOVEA	#data, An	1001 000 111 000 000	28	WLG	0	SUBA.L (Opt)
MOVEA	<ea0>, An	0000 000 000 000 000	26	WL		
MOVEC	Rc, Dn	0100 111 001 111 010	42	1PL		
MOVEC	Rc, An	0100 111 001 111 010	42	1PL		
MOVEC	Dn, Rc	0100 111 001 111 011	42	1PL		
MOVEC	An, Rc	0100 111 001 111 011	42	1PL		
MOVEM	<ea8>, rlist	0100 110 010 000 000	29	WL		
MOVEM	rlist, <ea4>	0100 100 010 000 000	30	WL		
MOVEP	d (An), Dn	0000 000 100 001 000	5	W		
MOVEP	Dn, d (An)	0000 000 110 001 000	5	W		
MOVEP	d (An), Dn	0000 000 101 001 000	5	L		
MOVEP	Dn, d (An)	0000 000 111 001 000	5	L		
MOVEQ	#data, Dn	0111 000 0 00000000	11	L	-8	
MOVEQ	#data, An	1001 000 111 000 000	28	LG	0	SUBA.L (Opt)
MOVES	Dn, <ea2>	0000 111 000 000 000	43	1PBWL		
MOVES	An, <ea2>	0000 111 000 000 000	43	1PBWL		
MOVES	<ea2>, Dn	0000 111 000 000 000	43	1PBWL		
MOVES	<ea2>, An	0000 111 000 000 000	43	1PBWL		
MULS	special	0100 110 000 000 000	38	2L		
MULS	<ea6>, Dn	1100 000 111 000 000	21	W		
MULU	special	0100 110 000 000 000	39	2L		
MULU	<ea6>, Dn	1100 000 011 000 000	21	W		
NBCD	<ea3>	0100 100 000 000 000	15	B		
NEG	<ea3>	0100 010 000 000 000	17	BWL		
NEGX	<ea3>	0100 000 000 000 000	17	BWL		
NOP		0100 111 001 110 001	0			

■ Table F-7 (continued)

MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
NOT	<ea3>	0100 011 000 000 000	17	BWL		
OR	#data, <ea3>	0000 000 000 000 000	25	BWLG		ORI
OR	<ea6>, Dn	1000 000 000 000 000	22	BWL		
OR	Dn, <ea2>	1000 000 100 000 000	23	BWL		
ORI	#data, <ea3>	0000 000 000 000 000	25	BWL		
ORI	#data, CCR	0000 000 000 111 100	1	B	8	
ORI	#data, SR	0000 000 001 111 100	1	PW	16	
PACK	special	1000 000 101 000 000	44	2	16	
PEA	<ea7>	0100 100 001 000 000	45	L		
RESET		0100 111 001 110 000	0	P		
ROL	#data, Dn	1110 000 100 011 000	10	BWL	3	
ROL	Dn, Dn	1110 000 100 111 000	9	BWL		
ROL	<ea2>	1110 011 011 000 000	15	W		
ROR	#data, Dn	1110 000 000 011 000	10	BWL	3	
ROR	Dn, Dn	1110 000 000 111 000	9	BWL		
ROR	<ea2>	1110 011 111 000 000	15	W		
ROXL	#data, Dn	1110 000 100 010 000	10	BWL	3	
ROXL	Dn, Dn	1110 000 100 110 000	9	BWL		
ROXL	<ea2>	1110 010 111 000 000	15	W		
ROXR	#data, Dn	1110 000 000 010 000	10	BWL	3	
ROXR	Dn, Dn	1110 000 000 110 000	9	BWL		
ROXR	<ea2>	1110 010 011 000 000	15	W		
RTD	#data	0100 111 001 110 100	1	1W	-16	
RTE		0100 111 001 110 011	0	P		
RTM	Dn	0000 011 011 000 000	48	2		
RTM	An	0000 011 011 001 000	48	2		
RTR		0100 111 001 110 111	0			
RTS		0100 111 001 110 101	0			
SBCD	Dn, Dn	1000 000 100 000 000	6	B		
SBCD	-(An), -(An)	1000 000 100 001 000	6	B		
SCC	<ea3>	0101 0100 11 000 000	15	B		
SCS	<ea3>	0101 0101 11 000 000	15	B		

(continued)

■ Table F-7 (continued)

MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
SEQ	<ea3>	0101 0111 11 000 000	15	B		
SF	<ea3>	0101 0001 11 000 000	15	B		
SGE	<ea3>	0101 1100 11 000 000	15	B		
SGT	<ea3>	0101 1110 11 000 000	15	B		
SHI	<ea3>	0101 0010 11 000 000	15	B		
SHS	<ea3>	0101 0100 11 000 000	15	BG		SCC
SLE	<ea3>	0101 1111 11 000 000	15	B		
SLO	<ea3>	0101 0101 11 000 000	15	BG		SCS
SLS	<ea3>	0101 0011 11 000 000	15	B		
SLT	<ea3>	0101 1101 11 000 000	15	B		
SMI	<ea3>	0101 1011 11 000 000	15	B		
SNE	<ea3>	0101 0110 11 000 000	15	B		
SNZ	<ea3>	0101 0110 11 000 000	15	BG		SNE
SPL	<ea3>	0101 1010 11 000 000	15	B		
ST	<ea3>	0101 0000 11 000 000	15	B		
STOP	#data	0100 111 001 110 010	1	P	16	
SUB	#data, <ea1>	0101 000 100 000 000	24	BWLG	3	SUBQ (Opt)
SUB	#data, <ea3>	0000 010 000 000 000	25	BWLG		SUBI
SUB	Dn, <ea2>	1001 000 100 000 000	23	BWL		
SUB	<ea0>, Dn	1001 000 000 000 000	22	BWL		
SUB	<ea0>, An	1001 000 011 000 000	27	WLG		SUBA
SUBA	#data, <ea1>	0101 000 100 000 000	24	WLG	3	SUBQ (Opt)
SUBA	<ea0>, An	1001 000 011 000 000	27	WL		
SUBI	#data, <ea3>	0000 010 000 000 000	25	BWL		
SUBQ	#data, <ea1>	0101 000 100 000 000	24	BWL	3	
SUBX	Dn, Dn	1001 000 100 000 000	8	BWL		
SUBX	-(An), -(An)	1001 000 100 001 000	8	BWL		
SVC	<ea3>	0101 1000 11 000 000	15	B		
SVS	<ea3>	0101 1001 11 000 000	15	B		
SWAP	Dn	0100 100 001 000 000	2	W		
TAS	<ea3>	0100 101 011 000 000	15	B		
TCC		0101 0100 11111 100	0	2		



■ Table F-7 (continued)

MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
TCS		0101 0101 11111 100	0	2		
TDIVS	<i>special</i>	0100 110 001 000 000	40	2L		
TDIVU	<i>special</i>	0100 110 001 000 000	41	2L		
TEQ		0101 0111 11111 100	0	2		
TF		0101 0001 11111 100	0	2		
TGE		0101 1100 11111 100	0	2		
TGT		0101 1110 11111 100	0	2		
THI		0101 0010 11111 100	0	2		
THS		0101 0100 11111 100	0	2G		TCC
TLE		0101 1111 11111 100	0	2		
TLO		0101 0101 11111 100	0	2G		TCS
TLS		0101 0011 11111 100	0	2		
TLT		0101 1101 11111 100	0	2		
TMI		0101 1011 11111 100	0	2		
TNE		0101 0110 11111 100	0	2		
TNZ		0101 0110 11111 100	0	2G		TNE
TPCC	<i>#data</i>	0101 0100 11111 010	1	2W	-16	
TPCC	<i>#data</i>	0101 0100 11111 011	1	2L		
TPCS	<i>#data</i>	0101 0101 11111 010	1	2W	-16	
TPCS	<i>#data</i>	0101 0101 11111 011	1	2L		
TPEQ	<i>#data</i>	0101 0111 11111 010	1	2W	-16	
TPEQ	<i>#data</i>	0101 0111 11111 011	1	2L		
TPF	<i>#data</i>	0101 0001 11111 010	1	2W	-16	
TPF	<i>#data</i>	0101 0001 11111 011	1	2L		
TPGE	<i>#data</i>	0101 1100 11111 010	1	2W	-16	
TPGE	<i>#data</i>	0101 1100 11111 011	1	2L		
TPGT	<i>#data</i>	0101 1110 11111 010	1	2W	-16	
TPGT	<i>#data</i>	0101 1110 11111 011	1	2L		
TPHI	<i>#data</i>	0101 0010 11111 010	1	2W	-16	
TPHI	<i>#data</i>	0101 0010 11111 011	1	2L		
TPHS	<i>#data</i>	0101 0100 11111 010	1	2WG	-16	TPCC
TPHS	<i>#data</i>	0101 0100 11111 011	1	2LG		TPCC

(continued)

■ Table F-7 (continued)

MC68000, MC68010, and MC68020/MC68030 instructions

Opcode	Operands	Opcode word	Group	Flags	Range	Equivalent
TPL		0101 1010 11111 100	0	2		
TPLE	#data	0101 1111 11111 010	1	2W	-16	
TPLE	#data	0101 1111 11111 011	1	2L		
TPLO	#data	0101 0101 11111 010	1	2WG	-16	TPCS
TPLO	#data	0101 0101 11111 011	1	2LG		TPCS
TPLS	#data	0101 0011 11111 010	1	2W	-16	
TPLS	#data	0101 0011 11111 011	1	2L		
TPLT	#data	0101 1101 11111 010	1	2W	-16	
TPLT	#data	0101 1101 11111 011	1	2L		
TPMI	#data	0101 1011 11111 010	1	2W	-16	
TPMI	#data	0101 1011 11111 011	1	2L		
TPNE	#data	0101 0110 11111 010	1	2W	-16	
TPNE	#data	0101 0110 11111 011	1	2L		
TPNZ	#data	0101 0110 11111 010	1	2WG	-16	TPNE
TPNZ	#data	0101 0110 11111 011	1	2LG		TPNE
TPPL	#data	0101 1010 11111 010	1	2W	-16	
TPPL	#data	0101 1010 11111 011	1	2L		
TPT	#data	0101 0000 11111 010	1	2W	-16	
TPT	#data	0101 0000 11111 011	1	2L		
TPVC	#data	0101 1000 11111 010	1	2W	-16	
TPVC	#data	0101 1000 11111 011	1	2L		
TPVS	#data	0101 1001 11111 010	1	2W	-16	
TPVS	#data	0101 1001 11111 011	1	2L		
TPZ	#data	0101 0111 11111 010	1	2W	-16	TPEQ
TPZ	#data	0101 0111 11111 011	1	2L		TPEQ
TRAP	#data	0100 111 001 00 0000	12		4	
TRAPV		0100 111 001 110 110	0			
TST	<ea3>	0100 101 000 000 000	17	BWL		
TT		0101 0000 11111 100	0	2		
TVC		0101 1000 11111 100	0	2		
TVS		0101 1001 11111 100	0	2		
TZ		0101 0111 11111 100	0	2		TEQ
UNLK	an	0100 111 001 011 000	2			
UNPK	special	1000 000 110 000 000	44	2	16	

■ Table F-8 MC68881 instructions

Opcode	Operands	Opcode word	Cp type	Group	Flags	Equivalent
FABS	FPn	000 000 000 0011000	Gen1	1	X	
FABS	FPn, FPn	000 000 000 0011000	Gen1	2	X	
FABS	<ea6>, FPn	010 000 000 0011000	Gen2	3	BWLSDXK	
FACOS	FPn	000 000 000 0011100	Gen1	1	X	
FACOS	FPn, FPn	000 000 000 0011100	Gen1	2	X	
FACOS	<ea6>, FPn	010 000 000 0011100	Gen2	3	BWLSDXK	
FADD	FPn	000 000 000 0100010	Gen1	1	XG	FADD FPn, FPn
FADD	FPn, FPn	000 000 000 0100010	Gen1	2	X	
FADD	<ea6>, FPn	010 000 000 0100010	Gen2	3	BWLSDXK	
FASIN	FPn	000 000 000 0001100	Gen1	1	X	
FASIN	FPn, FPn	000 000 000 0001100	Gen1	2	X	
FASIN	<ea6>, FPn	010 000 000 0001100	Gen2	3	BWLSDXK	
FATAN	FPn	000 000 000 0001010	Gen1	1	X	
FATAN	FPn, FPn	000 000 000 0001010	Gen1	2	X	
FATAN	<ea6>, FPn	010 000 000 0001010	Gen2	3	BWLSDXK	
FATANH	FPn	000 000 000 0001101	Gen1	1	X	
FATANH	FPn, FPn	000 000 000 0001101	Gen1	2	X	
FATANH	<ea6>, FPn	010 000 000 0001101	Gen2	3	BWLSDXK	
FBEQ	label	1111 000 01 0 000001	Bcc	4	WL	
FBF	label	1111 000 01 0 000000	Bcc	4	WL	
FBGE	label	1111 000 01 0 010011	Bcc	4	WL	
FBGL	label	1111 000 01 0 010110	Bcc	4	WL	
FBGLE	label	1111 000 01 0 010111	Bcc	4	WL	
FBGT	label	1111 000 01 0 010010	Bcc	4	WL	
FBLE	label	1111 000 01 0 010101	Bcc	4	WL	
FBLT	label	1111 000 01 0 010100	Bcc	4	WL	
FBNE	label	1111 000 01 0 001110	Bcc	4	WL	
FBNGE	label	1111 000 01 0 011100	Bcc	4	WL	
FBNGL	label	1111 000 01 0 011001	Bcc	4	WL	
FBNGLE	label	1111 000 01 0 011000	Bcc	4	WL	
FBNGT	label	1111 000 01 0 011101	Bcc	4	WL	
FBNLE	label	1111 000 01 0 011010	Bcc	4	WL	
FBNLT	label	1111 000 01 0 011011	Bcc	4	WL	
FBOGE	label	1111 000 01 0 000011	Bcc	4	WL	
FBOGL	label	1111 000 01 0 000110	Bcc	4	WL	
FBOGT	label	1111 000 01 0 000010	Bcc	4	WL	
FBOLE	label	1111 000 01 0 000101	Bcc	4	WL	
FBOLT	label	1111 000 01 0 000111	Bcc	4	WL	
FBRA	label	1111 000 01 0 001111	Bcc	4	WLGFBT	
FBSEQ	label	1111 000 01 0 010001	Bcc	4	WL	
FBSF	label	1111 000 01 0 010000	Bcc	4	WL	
FBSNE	label	1111 000 01 0 011110	Bcc	4	WL	
FBST	label	1111 000 01 0 011111	Bcc	4	WL	

(continued)

■ Table F-8 (continued)

MC68881 instructions

Opcode	Operands	Opcode word	Cp type	Group	Flags	Equivalent
FBT	<i>label</i>	1111 000 01 0 001111	Bcc	4	WL	
FBUEQ	<i>label</i>	1111 000 01 0 001001	Bcc	4	WL	
FBUGE	<i>label</i>	1111 000 01 0 001011	Bcc	4	WL	
FBUGT	<i>label</i>	1111 000 01 0 001010	Bcc	4	WL	
FBULE	<i>label</i>	1111 000 01 0 001101	Bcc	4	WL	
FBULT	<i>label</i>	1111 000 01 0 001100	Bcc	4	WL	
FBUN	<i>label</i>	1111 000 01 0 001000	Bcc	4	WL	
FCMP	<i>FPn, FPn</i>	000 000 000 0111000	Gen1	2	X	
FCMP	<i>&lt;ea6&gt;, FPn</i>	010 000 000 0111000	Gen2	3	BWLSDXK	
FCOS	<i>FPn</i>	000 000 000 0011101	Gen1	1	X	
FCOS	<i>FPn, FPn</i>	000 000 000 0011101	Gen1	2	X	
FCOS	<i>&lt;ea6&gt;, FPn</i>	010 000 000 0011101	Gen2	3	BWLSDXK	
FCOSH	<i>FPn</i>	000 000 000 0011001	Gen1	1	X	
FCOSH	<i>FPn, FPn</i>	000 000 000 0011001	Gen1	2	X	
FCOSH	<i>&lt;ea6&gt;, FPn</i>	010 000 000 0011001	Gen2	3	BWLSDXK	
FDBEQ	<i>Dn, label</i>	0000000000 000001	DBcc	5	W	
FDBF	<i>Dn, label</i>	0000000000 000000	DBcc	5	W	
FDBG	<i>Dn, label</i>	0000000000 010011	DBcc	5	W	
FDBGL	<i>Dn, label</i>	0000000000 010110	DBcc	5	W	
FDBGLE	<i>Dn, label</i>	0000000000 010111	DBcc	5	W	
FDBGT	<i>Dn, label</i>	0000000000 010010	DBcc	5	W	
FDBLE	<i>Dn, label</i>	0000000000 010101	DBcc	5	W	
FDBLT	<i>Dn, label</i>	0000000000 010100	DBcc	5	W	
FDBNE	<i>Dn, label</i>	0000000000 001110	DBcc	5	W	
FDBNGE	<i>Dn, label</i>	0000000000 011100	DBcc	5	W	
FDBNGL	<i>Dn, label</i>	0000000000 011001	DBcc	5	W	
FDBNGLE	<i>Dn, label</i>	0000000000 011000	DBcc	5	W	
FDBNGT	<i>Dn, label</i>	0000000000 011101	DBcc	5	W	
FDBNLE	<i>Dn, label</i>	0000000000 011010	DBcc	5	W	
FDBNLT	<i>Dn, label</i>	0000000000 011011	DBcc	5	W	
FDBOGE	<i>Dn, label</i>	0000000000 000011	DBcc	5	W	
FDBOGL	<i>Dn, label</i>	0000000000 000110	DBcc	5	W	
FDBOGT	<i>Dn, label</i>	0000000000 000010	DBcc	5	W	
FDBOLE	<i>Dn, label</i>	0000000000 000101	DBcc	5	W	
FDBOLT	<i>Dn, label</i>	0000000000 000100	DBcc	5	W	
FDBOR	<i>Dn, label</i>	0000000000 000111	DBcc	5	W	
FDBRA	<i>Dn, label</i>	0000000000 001111	DBcc	5	WGFDDBT	
FDBSEQ	<i>Dn, label</i>	0000000000 010001	DBcc	5	W	
FDBSF	<i>Dn, label</i>	0000000000 010000	DBcc	5	W	
FDBSNE	<i>Dn, label</i>	0000000000 011110	DBcc	5	W	
FDBST	<i>Dn, label</i>	0000000000 011111	DBcc	5	W	
FDBT	<i>Dn, label</i>	0000000000 001111	DBcc	5	W	
FDBUEQ	<i>Dn, label</i>	0000000000 001001	DBcc	5	W	
FDBUGE	<i>Dn, label</i>	0000000000 001011	DBcc	5	W	
FBUGT	<i>Dn, label</i>	0000000000 001010	DBcc	5	W	

■ Table F-8 (continued)

MC68881 instructions

Opcode	Operands	Opcode word	Cp type	Group	Flags	Equivalent
FDBULE	<i>Dn, label</i>	0000000000 001101	DBcc	5	W	
FDBULT	<i>Dn, label</i>	0000000000 001100	DBcc	5	W	
FDBUN	<i>Dn, label</i>	0000000000 001000	DBcc	5	W	
FDIV	<i>FPn, FPn</i>	000 000 000 01000000	Gen1	2	X	
FDIV	<i>&lt;ea6&gt;, FPn</i>	010 000 000 01000000	Gen2	3	BWLSDXK	
FETOX	<i>FPn</i>	000 000 000 00100000	Gen1	1	X	
FETOX	<i>FPn, FPn</i>	000 000 000 00100000	Gen1	2	X	
FETOX	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00100000	Gen2	3	BWLSDXK	
FETOXM1	<i>FPn</i>	000 000 000 00010000	Gen1	1	X	
FETOXM1	<i>FPn, FPn</i>	000 000 000 00010000	Gen1	2	X	
FETOXM1	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00010000	Gen2	3	BWLSDXK	
FGETEXP	<i>FPn</i>	000 000 000 00111100	Gen1	1	X	
FGETEXP	<i>FPn, FPn</i>	000 000 000 00111100	Gen1	2	X	
FGETEXP	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00111100	Gen2	3	BWLSDXK	
FGETMAN	<i>FPn</i>	000 000 000 00111111	Gen1	1	X	
FGETMAN	<i>FPn, FPn</i>	000 000 000 00111111	Gen1	2	X	
FGETMAN	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00111111	Gen2	3	BWLSDXK	
FINT	<i>FPn</i>	000 000 000 00000001	Gen1	1	X	
FINT	<i>FPn, FPn</i>	000 000 000 00000001	Gen1	2	X	
FINT	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00000001	Gen2	3	BWLSDXK	
FINTRZ	<i>FPn</i>	000 000 000 00000011	Gen1	1	X	
FINTRZ	<i>FPn, FPn</i>	000 000 000 00000011	Gen1	2	X	
FINTRZ	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00000011	Gen2	3	BWLSDXK	
FLOG10	<i>FPn</i>	000 000 000 00101010	Gen1	1	X	
FLOG10	<i>FPn, FPn</i>	000 000 000 00101010	Gen1	2	X	
FLOG10	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00101010	Gen2	3	BWLSDXK	
FLOG2	<i>FPn</i>	000 000 000 00101100	Gen1	1	X	
FLOG2	<i>FPn, FPn</i>	000 000 000 00101100	Gen1	2	X	
FLOG2	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00101100	Gen2	3	BWLSDXK	
FLOGN	<i>FPn</i>	000 000 000 00101000	Gen1	1	X	
FLOGN	<i>FPn, FPn</i>	000 000 000 00101000	Gen1	2	X	
FLOGN	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00101000	Gen2	3	BWLSDXK	
FLOGNP1	<i>FPn</i>	000 000 000 00000110	Gen1	1	X	
FLOGNP1	<i>FPn, FPn</i>	000 000 000 00000110	Gen1	2	X	
FLOGNP1	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00000110	Gen2	3	BWLSDXK	
FMOD	<i>FPn, FPn</i>	000 000 000 01000001	Gen1	2	X	
FMOD	<i>&lt;ea6&gt;, FPn</i>	010 000 000 01000001	Gen2	3	BWLSDXK	
FMOVE	<i>special</i>	011 000 000 00000000		21	P {k-factor}	
FMOVE	<i>FPn, FPn</i>	000 000 000 00000000	Gen1	2	X	
FMOVE	<i>FPn, &lt;ea3&gt;</i>	011 000 000 00000000	Gen3	6	BWLSDX	
FMOVE	<i>&lt;ea6&gt;, FPn</i>	010 000 000 00000000	Gen2	3	BWLSDXK	
FMOVE	<i>FPCR, &lt;ea3&gt;</i>	101 100 0000000000	Gen3	7	L	
FMOVE	<i>FPSR, &lt;ea3&gt;</i>	101 010 0000000000	Gen3	7	L	
FMOVE	<i>FPIAR, &lt;ea3&gt;</i>	101 001 0000000000	Gen3	7	L	

(continued)

■ Table F-8 (continued)

MC68881 instructions

Opcode	Operands	Opcode word	Cp type	Group	Flags	Equivalent
FMOVE	FPIAR, <i>An</i>	101 001 0000000000	Gen3	7	L	
FMOVE	< <i>ea6</i> >, FPCR	100 100 0000000000	Gen2	8	L	
FMOVE	< <i>ea6</i> >, FPSR	100 010 0000000000	Gen2	8	L	
FMOVE	< <i>ea6</i> >, FPIAR	100 001 0000000000	Gen2	8	L	
FMOVE	<i>An</i> , FPIAR	100 001 0000000000	Gen2	8	L	
FMOVECR	<i>#data</i> , FPN	010111 000 00000000	Gen1	9	X	
FMOVEM	< <i>ea8</i> >, <i>frlist</i>	110 10 000 0000000000	Gen2	10	X	
FMOVEM	<i>frlist</i> , < <i>ea4</i> >	111 00 000 0000000000	Gen3	11	X	
FMOVEM	< <i>ea8</i> >, <i>Dn</i>	110 11 000 0000000000	Gen2	12	X	
FMOVEM	<i>Dn</i> , < <i>ea4</i> >	111 01 000 0000000000	Gen3	13	X	
FMOVEM	< <i>ea0</i> >, <i>scrlist</i>	11 0 000 000000000000	Gen2	14	L	
FMOVEM	<i>scrlist</i> , < <i>ea1</i> >	11 1 000 000000000000	Gen3	15	L	
FMUL	FPN	000 000 000 0100011	Gen1	1	XG	FMUL FPN, FPN
FMUL	FPN, FPN	000 000 000 0100011	Gen1	2	X	
FMUL	< <i>ea6</i> >, FPN	010 000 000 0100011	Gen2	3		BWLSDXK
FNEG	FPN	000 000 000 0011010	Gen1	1	X	
FNEG	FPN, FPN	000 000 000 0011010	Gen1	2	X	
FNEG	< <i>ea6</i> >, FPN	010 000 000 0011010	Gen2	3		BWLSDXK
FNOP		000000000000000000		0		
FREM	FPN, FPN	000 000 000 0100101	Gen1	2	X	
FREM	< <i>ea6</i> >, FPN	010 000 000 0100101	Gen2	3		BWLSDXK
FRESTORE	< <i>ea8</i> >	1111 000 101 000 000	Rest	16	P	
FSAVE	< <i>ea4</i> >	1111 000 100 000 000	Save	16	P	
FSCALE	FPN, FPN	000 000 000 0100110	Gen1	2	X	
FSCALE	< <i>ea6</i> >, FPN	010 000 000 0100110	Gen2	3		BWLSDXK
FSEQ	< <i>ea3</i> >	0000000000 000001	ScC	17	B	
FSF	< <i>ea3</i> >	0000000000 000000	ScC	17	B	
FSGE	< <i>ea3</i> >	0000000000 010011	ScC	17	B	
FSGL	< <i>ea3</i> >	0000000000 010110	ScC	17	B	
FSGLE	< <i>ea3</i> >	0000000000 010111	ScC	17	B	
FSGT	< <i>ea3</i> >	0000000000 010010	ScC	17	B	
FSLE	< <i>ea3</i> >	0000000000 010101	ScC	17	B	
FSLT	< <i>ea3</i> >	0000000000 010100	ScC	17	B	
FSNE	< <i>ea3</i> >	0000000000 001110	ScC	17	B	
FSNGE	< <i>ea3</i> >	0000000000 011100	ScC	17	B	
FSNGL	< <i>ea3</i> >	0000000000 011001	ScC	17	B	
FSNGLE	< <i>ea3</i> >	0000000000 011000	ScC	17	B	
FSNGT	< <i>ea3</i> >	0000000000 011101	ScC	17	B	
FSNLE	< <i>ea3</i> >	0000000000 011010	ScC	17	B	
FSNLT	< <i>ea3</i> >	0000000000 011011	ScC	17	B	
FSGE	< <i>ea3</i> >	0000000000 000011	ScC	17	B	
FSGL	< <i>ea3</i> >	0000000000 000110	ScC	17	B	
FSGT	< <i>ea3</i> >	0000000000 000010	ScC	17	B	
FSOLE	< <i>ea3</i> >	0000000000 000101	ScC	17	B	
FSOLT	< <i>ea3</i> >	0000000000 000100	ScC	17	B	

■ Table F-8 (continued)

MC68881 instructions

Opcode	Operands	Opcode word	Cp type	Group	Flags	Equivalent
FSOR	<ea3>	0000000000 000111	Scc	17	B	
FSSEQ	<ea3>	0000000000 010001	Scc	17	B	
FSSF	<ea3>	0000000000 010000	Scc	17	B	
FSSNE	<ea3>	0000000000 011110	Scc	17	B	
FSST	<ea3>	0000000000 011111	Scc	17	B	
FST	<ea3>	0000000000 001111	Scc	17	B	
FSUEQ	<ea3>	0000000000 001001	Scc	17	B	
FSUGE	<ea3>	0000000000 001011	Scc	17	B	
FSUGT	<ea3>	0000000000 001010	Scc	17	B	
FSULE	<ea3>	0000000000 001101	Scc	17	B	
FSULT	<ea3>	0000000000 001100	Scc	17	B	
FSUN	<ea3>	0000000000 001000	Scc	17	B	
FSGLDIV	FPn, FPn	000 000 000 0100100	Gen1	2	X	
FSGLDIV	<ea6>, FPn	010 000 000 0100100	Gen2	3	BWLSDXK	
FSGLMUL	FPn, FPn	000 000 000 0100111	Gen1	2	X	
FSGLMUL	<ea6>, FPn	010 000 000 0100111	Gen2	3	BWLSDXK	
FSIN	FPn	000 000 000 0001110	Gen1	1	X	
FSIN	FPn, FPn	000 000 000 0001110	Gen1	2	X	
FSIN	<ea6>, FPn	010 000 000 0001110	Gen2	3	BWLSDXK	
FSINCOS	special	000 000 000 0110 000		22	BWLSDXK	
FSINH	FPn	000 000 000 0000010	Gen1	1	X	
FSINH	FPn, FPn	000 000 000 0000010	Gen1	2	X	
FSINH	<ea6>, FPn	010 000 000 0000010	Gen2	3	BWLSDXK	
FSQRT	FPn	000 000 000 0000100	Gen1	1	X	
FSQRT	FPn, FPn	000 000 000 0000100	Gen1	2	X	
FSQRT	<ea6>, FPn	010 000 000 0000100	Gen2	3	BWLSDXK	
FSUB	FPn, FPn	000 000 000 0101000	Gen1	2	X	
FSUB	<ea6>, FPn	010 000 000 0101000	Gen2	3	BWLSDXK	
FTAN	FPn	000 000 000 0001111	Gen1	1	X	
FTAN	FPn, FPn	000 000 000 0001111	Gen1	2	X	
FTAN	<ea6>, FPn	010 000 000 0001111	Gen2	3	BWLSDXK	
FTANH	FPn	000 000 000 0001001	Gen1	1	X	
FTANH	FPn, FPn	000 000 000 0001001	Gen1	2	X	
FTANH	<ea6>, FPn	010 000 000 0001001	Gen2	3	BWLSDXK	
FTENTOX	FPn	000 000 000 0010010	Gen1	1	X	
FTENTOX	FPn, FPn	000 000 000 0010010	Gen1	2	X	
FTENTOX	<ea6>, FPn	010 000 000 0010010	Gen2	3	BWLSDXK	
FTEST	FPn	000 000 000 0111010	Gen1	20	X	
FTEST	<ea6>	010 000 000 0111010	Gen2	20	BWLSDXK	
FTEQ		0000000000 000001	Tcc1	18		
FTF		0000000000 000000	Tcc1	18		
FTGE		0000000000 010011	Tcc1	18		
FTGL		0000000000 010110	Tcc1	18		
FTGLE		0000000000 010111	Tcc1	18		

(continued)

■ Table F-8 (continued)

MC68881 instructions

Opcode	Operands	Opcode word	Cp type	Group	Flags	Equivalent
FTGT		0000000000 010010	Tcc1	18		
FTLE		0000000000 010101	Tcc1	18		
FTLT		0000000000 010100	Tcc1	18		
FTNE		0000000000 001110	Tcc1	18		
FTNGE		0000000000 011100	Tcc1	18		
FTNGL		0000000000 011001	Tcc1	18		
FTNGLE		0000000000 011000	Tcc1	18		
FTNGT		0000000000 011101	Tcc1	18		
FTNLE		0000000000 011010	Tcc1	18		
FTNLT		0000000000 011011	Tcc1	18		
FTOGE		0000000000 000011	Tcc1	18		
FTOGL		0000000000 000110	Tcc1	18		
FTOGT		0000000000 000010	Tcc1	18		
FTOLE		0000000000 000101	Tcc1	18		
FTOLT		0000000000 000100	Tcc1	18		
FTOR		0000000000 000111	Tcc1	18		
FTSEQ		0000000000 010001	Tcc1	18		
FTSF		0000000000 010000	Tcc1	18		
FTSNE		0000000000 011110	Tcc1	18		
FTST		0000000000 011111	Tcc1	18		
FTT		0000000000 001111	Tcc1	18		
FTUEQ		0000000000 001001	Tcc1	18		
FTUGE		0000000000 001011	Tcc1	18		
FTUGT		0000000000 001010	Tcc1	18		
FTULE		0000000000 001101	Tcc1	18		
FTULT		0000000000 001100	Tcc1	18		
FTUN		0000000000 001000	Tcc1	18		
FTPEQ	# data	0000000000 000001	Tcc2	19	WL	
FTPF	# data	0000000000 000000	Tcc2	19	WL	
FTPGE	# data	0000000000 010011	Tcc2	19	WL	
FTPGL	# data	0000000000 010110	Tcc2	19	WL	
FTPGLE	# data	0000000000 010111	Tcc2	19	WL	
FTPGT	# data	0000000000 010010	Tcc2	19	WL	
FTPLE	# data	0000000000 010101	Tcc2	19	WL	
FTPLT	# data	0000000000 010100	Tcc2	19	WL	
FTPNE	# data	0000000000 001110	Tcc2	19	WL	
FTPNGE	# data	0000000000 011100	Tcc2	19	WL	
FTPNGL	# data	0000000000 011001	Tcc2	19	WL	
FTPNGLE	# data	0000000000 011000	Tcc2	19	WL	
FTPNGT	# data	0000000000 011101	Tcc2	19	WL	
FTPNLE	# data	0000000000 011010	Tcc2	19	WL	
FTPNLT	# data	0000000000 011011	Tcc2	19	WL	
FTPOGE	# data	0000000000 000011	Tcc2	19	WL	
FTPOGL	# data	0000000000 000110	Tcc2	19	WL	
FTPOGT	# data	0000000000 000010	Tcc2	19	WL	



■ Table F-8 (continued) MC68881 instructions

Opcode	Operands	Opcode word	Cp type	Group	Flags	Equivalent
FTPOLE	<i>#data</i>	0000000000 000101	Tcc2	19	WL	
FTPOLT	<i>#data</i>	0000000000 000100	Tcc2	19	WL	
FTPOR	<i>#data</i>	0000000000 000111	Tcc2	19	WL	
FTPSEQ	<i>#data</i>	0000000000 010001	Tcc2	19	WL	
FTPSF	<i>#data</i>	0000000000 010000	Tcc2	19	WL	
FTPSNE	<i>#data</i>	0000000000 011110	Tcc2	19	WL	
FTPST	<i>#data</i>	0000000000 011111	Tcc2	19	WL	
FTPT	<i>#data</i>	0000000000 001111	Tcc2	19	WL	
FTPUEQ	<i>#data</i>	0000000000 001001	Tcc2	19	WL	
FTPUGE	<i>#data</i>	0000000000 001011	Tcc2	19	WL	
FTPUGT	<i>#data</i>	0000000000 001010	Tcc2	19	WL	
FTPULE	<i>#data</i>	0000000000 001101	Tcc2	19	WL	
FTPULT	<i>#data</i>	0000000000 001100	Tcc2	19	WL	
FTPUN	<i>#data</i>	0000000000 001000	Tcc2	19	WL	
FTWOTOX	<i>FPn</i>	000 000 000 0010001	Gen1	1	X	
FTWOTOX	<i>FPn, FPn</i>	000 000 000 0010001	Gen1	2	X	
FTWOTOX	<i>&lt;ea6&gt;, FPn</i>	010 000 000 0010001	Gen2	3	BWLSDXK	

■ Table F-9 MC68851 instructions

Opcode	Operands	Opcode Word	Cp Type	Group	Flags
PBAS	<i>label</i>	1111 000 01 0 000110	Bcc	4	
PBAC	<i>label</i>	1111 000 01 0 000111	Bcc	4	
PBBS	<i>label</i>	1111 000 01 0 000000	Bcc	4	
PBBC	<i>label</i>	1111 000 01 0 000001	Bcc	4	
PBCS	<i>label</i>	1111 000 01 0 001110	Bcc	4	
PBCC	<i>label</i>	1111 000 01 0 001111	Bcc	4	
PBGS	<i>label</i>	1111 000 01 0 001100	Bcc	4	
PBGC	<i>label</i>	1111 000 01 0 001101	Bcc	4	
PBIS	<i>label</i>	1111 000 01 0 001010	Bcc	4	
PBIC	<i>label</i>	1111 000 01 0 001011	Bcc	4	
PBLS	<i>label</i>	1111 000 01 0 000010	Bcc	4	
PBLC	<i>label</i>	1111 000 01 0 000011	Bcc	4	
PBSS	<i>label</i>	1111 000 01 0 000100	Bcc	4	
PBSC	<i>label</i>	1111 000 01 0 000101	Bcc	4	
PBWS	<i>label</i>	1111 000 01 0 001000	Bcc	4	
PBWC	<i>label</i>	1111 000 01 0 001001	Bcc	4	
PDBAS	<i>Dn, label</i>	0000000000 000110	DBcc	5	
PDBAC	<i>Dn, label</i>	0000000000 000111	DBcc	5	
PDBBS	<i>Dn, label</i>	0000000000 000000	DBcc	5	
PDBBC	<i>Dn, label</i>	0000000000 000001	DBcc	5	
PDBCS	<i>Dn, label</i>	0000000000 001110	DBcc	5	

(continued)

■ Table F-9 (continued)

MC68851 instructions

Opcode	Operands	Opcode Word	Cp Type	Group	Flags
PDBCC	<i>Dn, label</i>	0000000000 001111	DBcc	5	
PDBGS	<i>Dn, label</i>	0000000000 001100	DBcc	5	
PDBGC	<i>Dn, label</i>	0000000000 001101	DBcc	5	
PDBIS	<i>Dn, label</i>	0000000000 001010	DBcc	5	
PDBIC	<i>Dn, label</i>	0000000000 001011	DBcc	5	
PDBLS	<i>Dn, label</i>	0000000000 000010	DBcc	5	
PDBLC	<i>Dn, label</i>	0000000000 000011	DBcc	5	
PDBSS	<i>Dn, label</i>	0000000000 000100	DBcc	5	
PDBSC	<i>Dn, label</i>	0000000000 000101	DBcc	5	
PDEWS	<i>Dn, label</i>	0000000000 001000	DBcc	5	
PDEWC	<i>Dn, label</i>	0000000000 001001	DBcc	5	
PFLUSH	<i>special</i>	001 100 0 0000 00000		10	
PFLUSHA		001 001 0 0000 00000	Gen1	0	
PFLUSHR	<i>&lt;ea12&gt;</i>	101 00000000000000	Gen2	2	D
PFLUSHS	<i>special</i>	001 101 0 0000 00000		10	
PLOADR	<i>special</i>	001 000 1 0000 00000		11	
PLOADW	<i>special</i>	001 000 0 0000 00000		11	
PMOVE	<i>&lt;ea0&gt;, BADn</i>	011 000 0 00000000 00	Gen2	12	W
PMOVE	<i>BADn, &lt;ea1&gt;</i>	011 000 1 00000000 00	Gen3	12	W
PMOVE	<i>&lt;ea0&gt;, BACn</i>	011 000 0 00000000 00	Gen2	12	W
PMOVE	<i>BACn, &lt;ea1&gt;</i>	011 000 1 00000000 00	Gen3	12	W
PMOVE	<i>&lt;ea0&gt;, PSR</i>	011 000 0 00000000 00	Gen2	12	W
PMOVE	<i>PSR, &lt;ea1&gt;</i>	011 000 1 00000000 00	Gen3	12	W
PMOVE	<i>&lt;ea0&gt;, TC</i>	010 000 0 0000000000	Gen2	12	L
PMOVE	<i>TC, &lt;ea1&gt;</i>	010 000 1 0000000000	Gen3	12	L
PMOVE	<i>&lt;ea0&gt;, XRP</i>	010 000 0 0000000000	Gen2	12	D
PMOVE	<i>XRP, &lt;ea1&gt;</i>	010 000 1 0000000000	Gen3	12	D
PMOVE	<i>&lt;ea0&gt;, SCCCAL</i>	010 000 0 0000000000	Gen2	12	B
PMOVE	<i>SCCCAL, &lt;ea1&gt;</i>	010 000 1 0000000000	Gen3	12	B
PMOVE	<i>&lt;ea0&gt;, VAL</i>	010 000 0 0000000000	Gen2	12	B
PMOVE	<i>VAL, &lt;ea1&gt;</i>	010 000 1 0000000000	Gen3	12	B
PMOVE	<i>&lt;ea0&gt;, AC</i>	010 000 0 0000000000	Gen2	12	W
PMOVE	<i>AC, &lt;ea1&gt;</i>	010 000 1 0000000000	Gen3	12	W
PMOVE	<i>PCSR, &lt;ea1&gt;</i>	011 000 1 00000000 00	Gen3	12	W
PRESTORE	<i>&lt;ea8&gt;</i>	1111 000 101 000 000	Rest	6	
PSAVE	<i>&lt;ea4&gt;</i>	1111 000 100 000 000	Save	6	
PSAS	<i>&lt;ea3&gt;</i>	0000000000 000110	Scc	7	B
PSAC	<i>&lt;ea3&gt;</i>	0000000000 000111	Scc	7	B
PSBS	<i>&lt;ea3&gt;</i>	0000000000 000000	Scc	7	B
PSBC	<i>&lt;ea3&gt;</i>	0000000000 000001	Scc	7	B
PSCS	<i>&lt;ea3&gt;</i>	0000000000 001110	Scc	7	B
PSCC	<i>&lt;ea3&gt;</i>	0000000000 001111	Scc	7	B
PSGS	<i>&lt;ea3&gt;</i>	0000000000 001100	Scc	7	B
PSGC	<i>&lt;ea3&gt;</i>	0000000000 001101	Scc	7	B

■ Table F-9 (continued)

MC68851 instructions

Opcode	Operands	Opcode Word	Cp Type	Group	Flags
PSIS	<ea3>	0000000000 001010	Sccl	7	B
PSIC	<ea3>	0000000000 001011	Sccl	7	B
PSLS	<ea3>	0000000000 000010	Sccl	7	B
PSLC	<ea3>	0000000000 000011	Sccl	7	B
PSSS	<ea3>	0000000000 000100	Sccl	7	B
PSSC	<ea3>	0000000000 000101	Sccl	7	B
PSWS	<ea3>	0000000000 001000	Sccl	7	B
PSWC	<ea3>	0000000000 001001	Sccl	7	B
PTESTR	<i>special</i>	100 000 1 0000 00000		13	
PTESTW	<i>special</i>	100 000 0 0000 00000		13	
PTAS		0000000000 000110	Tcc1	8	
PTAC		0000000000 000111	Tcc1	8	
PTBS		0000000000 000000	Tcc1	8	
PTBC		0000000000 000001	Tcc1	8	
PTCS		0000000000 001110	Tcc1	8	
PTCC		0000000000 001111	Tcc1	8	
PTGS		0000000000 001100	Tcc1	8	
PTGC		0000000000 001101	Tcc1	8	
PTIS		0000000000 001010	Tcc1	8	
PTIC		0000000000 001011	Tcc1	8	
PTLS		0000000000 000010	Tcc1	8	
PTLC		0000000000 000011	Tcc1	8	
PTSS		0000000000 000100	Tcc1	8	
PTSC		0000000000 000101	Tcc1	8	
PTWS		0000000000 001000	Tcc1	8	
PTWC		0000000000 001001	Tcc1	8	
PTPAS	<i># data</i>	0000000000 000110	Tcc2	9	WL
PTPAC	<i># data</i>	0000000000 000111	Tcc2	9	WL
PTPBS	<i># data</i>	0000000000 000000	Tcc2	9	WL
PTPBC	<i># data</i>	0000000000 000001	Tcc2	9	WL
PTPCS	<i># data</i>	0000000000 001110	Tcc2	9	WL
PTPCC	<i># data</i>	0000000000 001111	Tcc2	9	WL
PTPGS	<i># data</i>	0000000000 001100	Tcc2	9	WL
PTPGC	<i># data</i>	0000000000 001101	Tcc2	9	WL
PTPIS	<i># data</i>	0000000000 001010	Tcc2	9	WL
PTPIC	<i># data</i>	0000000000 001011	Tcc2	9	WL
PTPLS	<i># data</i>	0000000000 000010	Tcc2	9	WL
PTPLC	<i># data</i>	0000000000 000011	Tcc2	9	WL
PTPSS	<i># data</i>	0000000000 000100	Tcc2	9	WL
PTPSC	<i># data</i>	0000000000 000101	Tcc2	9	WL
PTPWS	<i># data</i>	0000000000 001000	Tcc2	9	WL
PTPWC	<i># data</i>	0000000000 001001	Tcc2	9	WL
PVALID	VAL, <ea11>	001 010 0000000000	Gen3	3	L
PVALID	An, <ea11>	001 011 0000000000	Gen3	3	L



## Appendix G Assembler Command Syntax

THIS APPENDIX DEFINES THE COMMAND SYNTAX ACCEPTED BY the .MPW Assembler. It includes a detailed listing of the invocation options. ■



---

## Assembler command syntax

<b>Syntax</b>	<code>asm[option ...][file ...]</code>
<b>Description</b>	Assembles the specified assembly-language source files. One or more filenames may be specified. If no filenames are specified, standard input is assembled and the file <code>a.o</code> is created. By convention, assembly-language source filenames end in the suffix <code>.a</code> . Each file is assembled separately—assembling file <code>Name.a</code> creates object file <code>Name.a.o</code> . The object filename can be changed with the <code>-o</code> option.
<b>Input</b>	If no filenames are specified, standard input is assembled. (End-of-file is indicated by typing Command-Enter.)
<b>Output</b>	If either the <code>-l</code> or <code>-s</code> option is specified, an assembler listing is generated. If standard input is used for the source file, the listing is written to standard output. If the input is taken from file <code>Name.a</code> , the listing is written to <code>Name.a.lst</code> . The listing filename can be changed with the <code>-lo</code> option.
<b>Diagnostics</b>	Errors and warnings are written to diagnostic output. If the <code>-p</code> option is specified, progress and summary information is also written to diagnostic output.
<b>Status</b>	The following status values are returned to the Shell: <ul style="list-style-type: none"><li>0 No errors detected in any of the files assembled</li><li>1 Parameter or option errors</li><li>2 Errors detected</li><li>3 Execution terminated</li></ul>
<b>Options</b>	Except for the <code>-case on</code> option, options may appear in any order.  <b>-addrsize <i>size</i></b> Set address displays in the listing to <i>size</i> digits (values 4 to 8 are allowed). The default is 5 digits.  <b>-blksize <i>blocks</i></b> Set the Assembler's text file I/O buffer size to <i>blocks</i> * 512 bytes. Values 6 to 62 are allowed. Odd values are made even by reducing the value by 1. The default value is 16 (8192 bytes) if the Assembler determines it has the memory space for the I/O buffers, and 6 (3072 bytes) otherwise. This option permits optimization of I/O performance (transfer rate for text file input, load/dump files, and listing output) as a function to the disk device being used. Note that increasing the <i>blocks</i> value reduces the amount of memory available for other Assembler structures (such as symbol tables).

**-case on**

Distinguish between uppercase and lowercase letters in non-macro names (same as `CASE ON`). (Case is always ignored in macro names.) If you intend to preserve the case of names declared by the `-define` option, then the `case on` option must *precede* the `-define` option(s) in the command parameter list.

**-case obj[ect]**

Preserve the case of module, EXPORT, IMPORT, and ENTRY names *only in the generated object file*. In all other respects, case is ignored within the assembly, and the behavior is the same as the preset `CASE OFF` situation.

**-case off**

Ignore the case of letters. All identifiers are case insensitive. This is the preset mode of the Assembler, but it may be used in the command line to reverse the effect of one of the other `-case` modes.

**-[check]**

Syntax check only. No object file is generated.

**-d[efine] name [=value] [, name [=value]] ...**

Define the name as having the specified value. The value is a decimal integer. If *value* is omitted, a value of 1 is assumed. This option is equivalent to placing the directive

`name EQU value`

at the beginning of your source file. Note that in order to test whether or not the name is defined, the `&Type` function should be used. You can define more than one name by specifying multiple `-d` options or multiple `name [=value]` parameters separated by commas, as in this example:

`Asm -d debug1, &debug='on'...`

**-d[efine] &name [=value] [, &name [=value]] ...**

Define the macro *name* as having the specified value. The value is a decimal integer or a string constant. If the `"=value"` is omitted, the decimal value 1 is assumed. If only the *value* is omitted, the null string is assumed. `-define` is equivalent to declaring the name as a global arithmetic symbol (`GBLA` for an integer value) or global character macro symbol (`GBLC` for a string value) and placing one of the following directives at the beginning of the source file:

```
        GBLA &name
&name SETA value
or
        GBLC &name
&name SETC value
```



Note that in order to test whether the name is defined, the **&Type** function should be used. You can define more than one macro name by specifying multiple **-d** options or multiple **&name=value** parameters separated by commas.

**-e[errlog] filename**

Write all errors and warnings to the error log file with the specified filename (same as **ERRLOG 'filename'**). If only warnings are generated during assembly, the error log file is not created. Use of this option is discouraged.

**-f**

Suppress page ejects (same as **PRINT NOPAGE**).

**-font fontname[, fontsize]**

Set the listing font to *fontname* (for example, Courier), and the size to *fontsize*. This option is meaningful only if the **-s** or the **-I** option is used. The default listing font is Monaco 7. Note that listings will be formatted correctly only if a monospaced font is used.

**-h**

Suppress page headers (same as **PRINT NOHDR**).

**-i pathname [, pathname]...**

Search for include and load files in the specified directories. Multiple **-i** options may be specified. At most 15 directories will be searched. The search order is as follows:

1. The include or load filename is used as specified. If a *full pathname* is given, then no other searching is applied.

If the file wasn't found, and the pathname used to specify the file was a *partial pathname* (no colons in the name or a leading colon), then the following directories are searched.

2. The directory containing the current input file.
3. The directories specified in **-i** options, in the order listed.
4. The directories specified in the Shell variable **{AIncludes}**.

**-I**

Generate full listing. If file **Name.a** is assembled, the listing is written to **Name.a.lst**.

**-lo *listingname***

Pathname for the listing file and directory for the listing scratch file. If *listingname* ends with a colon (:), it indicates a directory for the listing file, whose name is then formed by the normal rules (that is, *inputFilename.a.lst*). If *listingname* does not end with a colon, the listing file is written to the file *listingname*. In this case, listings for multiple source files are appended to the listing file. In either case, the directory implied by the listing name is used for the assembler's listing scratch file. The **-lo** option is only meaningful if the **-s** or the **-I** option is used.

**-o *objname***

Pathname for the generated object file. If *objname* ends with a colon (:), it indicates a directory for the output file, whose name is then formed by the normal rules (that is, *inputFilename.o*). If *objname* does not end with a colon, the object file is written to the file *objname*. (In this case, only one source file should be specified to the Assembler.)

**-pagesize *l*, *w***

Set the listing page size. (This option is only meaningful if the **-s** or **-I** option is specified.) The *l* and *w* parameters are integers: *l* is the page length (default = 75) and *w* is the page width (default = 126). (These settings assume that Courier 7 is being used with the MPW Print command to the LaserWriter.)

**-P**

Write assembly progress information (module names, included, loads, and dumps) and summary information (number of errors, warnings, and compilation time) to the diagnostic output file. (This option is the same as **PRINT STAT**.)

**-print *mode* [, *mode*]...**

Set a print option mode. *Mode* may be any one of the following **PRINT** directive options:

[NO]DATA	Data
[NO]GEN	Macro expansions
[NO]HDR	Page headings
[NO]LITS	Literals
[NO]MCALL	Macro calls
[NO]MDIR	Macro directives
[NO]OBJ	Object code
[NO]PAGE	Page ejects
[NO]STAT	Progress information
[NO]SYM	Symbol table display
[NO]WARN	Warnings

See Chapter 4 for a discussion of these `PRINT` settings. You can specify more than one print option by specifying multiple `-print` options or multiple *mode* parameters separated by commas, as in this example:

```
Asm -print nowarn,noobj,nopage...
```

Note that single-letter options are provided for some of the settings: `-f` (`NOPAGE`), `-h` (`NOHDR`), `-p` (`STAT`), and `-w` (`NOWARN`).

**-s**

Set `PRINT NOOBJ` to generate a shortened form of the listing file. If the `-l` option is also specified, the rightmost option takes precedence.

**-sym off**

Do not write object file records containing information for SADE, the MPW symbolic debugger. This is the default, and will be in effect if no `sym` option is specified.

**-sym [on | full]**

Write complete object file records containing information for use by SADE. The options `on` and `full` are equivalent. The symbolic information generated by the assembler consists of Module Begin (entry) OMF records for identifiers defined by the `PROC`, `FUNC`, and `MAIN` directives, Local Identifier OMF records for all `EQU` and `SET` identifiers except for those identifiers defined in the files included from the `{AIncludes}` folder, and Local Label OMF records for the local code labels.

**-t**

Display the assembly time and the number of lines to the diagnostic file even if progress information (`-p`) is not being displayed.

**-w**

Suppress warning messages (same as `PRINT NOWARN`).

**-wb**

Suppress branch warning messages only.

### Example

```
Asm -w -l Sample.a Memory.a -d Debug
```

Assembles `Sample.a` and `Memory.a`, producing object files `Sample.a.o` and `Memory.a.o`. Suppresses warnings and defines the name "Debug" as having the value 1. Two listing files are generated: `Sample.a.lst` and `Memory.a.lst`. (These programs are located in the `AExamples` directory.)



## Appendix H Object Assembler Macros

THE FILE OBJMACROS.A CONTAINS A COLLECTION OF MPW ASSEMBLY-language macros. The macros permit you to write applications, or parts of applications, in an object-oriented fashion using assembly language.

Object-oriented assembly language allows you to

- create descendants of objects defined in Object Pascal
- call methods written in Object Pascal from assembly-language code
- call methods written in assembly language from Object Pascal

Object-oriented assembly language is as much a style of programming as a language. This chapter describes the macros provided for use with assembly language. For more complete information about object-oriented programming in general, see the *Macintosh Programmer's Workshop Pascal Reference*. For a sample program that uses object-oriented macros, see the *MacApp Programmer's Guide*. ■

### **Contents**

InitObjects	263
ObjectDef	263
ObjectIntf and the IMPL keyword	265
ObjectWith and EndObjectWith	266
ProcMethOf, FuncMethOf, and EndMethod	267
MethCall	268
Inherited	268
NewObject	269
MoveSelf	269



---

## InitObjects

`InitObjects` is a macro that must be called at the beginning of every program written entirely in assembly language using objects. In other words, this must be the first line of every object-oriented assembly-language program that isn't linked with an Object Pascal program.

The `InitObjects` prototype statement is

```
InitObjects
```

---

## ObjectDef

The `ObjectDef` macro allows you to define objects in object-oriented assembly-language programs. The `ObjectDef` macro also generates code so method calls are handled properly.

The `ObjectDef` macro sets up assembly language definitions so that you can later refer to fields of objects by using `%Typename` and the field name. For example, imagine that an object has this definition:

```
ObjectDef                                TArc, TShape,
\
    (nextShape, TShape),                  \
    (arcAngle, 2),                        \
    METHODS,                              \
    (Draw)
```

You can then use a statement like this:

```
MOVE.W    %TArc.arcAngle(A0), -(SP)
```

A percent sign (%) is appended to the object type identifier (other than when used in the `ObjectDef` and `ObjectWith` macros) so that you can use the Object Pascal object-type identifier in an `EQU` statement to define the size of an object and then use that name in place of the size for a field that refers to an object of that type. For example,

```
TShape    EQU    4
```

was used to set the value of the identifier `TShape` for the above `ObjectDef`.

- ◆ *Note:* Because an object reference is a handle, the `EQU` value will always be 4.

In object-oriented assembly language you must qualify the field name by the name of the object type that defined it. For example, if `TArc` inherits the field `fColor` from `TShape`, `%TArC.fColor` would be undefined. You would have to call it `%TShape.fColor`. The `ObjectWith` macro (described later in this appendix) is provided to get around this problem.

The `ObjectDef` prototype statement is

```
ObjectDef    &TypeName, &Heritage[, fieldList] [, METHODS, methodList]
```

- ◆ *Note:* The part of the prototype statement for `ObjectDef` that is in square brackets is optional. *FieldList* should be replaced by a list of all fields of the object type, with a size for each field, all in parentheses, as shown in the examples. *MethodList*, similarly, should be replaced by a list of the methods of the object type. If you have a method list, the word `METHODS` must be present. Otherwise, it should never be present.

These are some examples of the use of `ObjectDef`:

```
ObjectDef Shape,          TObject,    \
    (boundRect, 8),      \
    (borderThickness, 2), \
    (color, 2),          \
    METHODS,             \
    (Draw),              \
    (MoveBy),           \
    (Stretch)
ObjectDef Arc, Shape,    \
    (startAngle, 2),     \
    (arcAngle, 2),      \
    METHODS,             \
    (Draw, OVERRIDE),   \
    (GetArea),          \
    (SetArcAngle)
```

The numbers given after the field identifiers give the size, in bytes, of the storage required. If a method is inherited and reimplemented, you must qualify the method name with the word `OVERRIDE` as shown for the `Draw` method.



---

## ObjectIntf and the IMPL keyword

`ObjectIntf` is used to create an interface in assembly language for an object type that is declared in Object Pascal. It is the same as `ObjectDef` except that it does not generate method tables.

This macro is generally used when you want an assembly-language implementation of a method declared in an Object Pascal unit. It allows you to specify which methods will be implemented in assembly language. In the Object Pascal unit, you should declare the method `EXTERNAL`, as shown here:

```
Tfoo = OBJECT(TObject)
    field1: INTEGER;
    PROCEDURE Tfoo.Meth1;
    PROCEDURE Tfoo.Meth2;
END;

IMPLEMENTATION
    PROCEDURE Tfoo.Meth1; EXTERNAL;
    PROCEDURE Tfoo.Meth2;

BEGIN
- - -
END;
```

In the assembly-language file, you need to supply an `ObjectIntf` template for the class. You must give the entire object-type declaration in the `ObjectIntf` template. You give the `IMPL` keyword, preceded by a comma, after any method you want to implement in assembly language. The corresponding assembly-language declaration for the Object Pascal declaration just given is presented in "Examples," later in this appendix.

If the method for which you are providing an assembly-language implementation is a reimplement of an inherited method, you must specify both `OVERRIDE` and `IMPL`:

```
ObjectIntf                               MyShape, Shape
\
- - -                                     \
(Draw, OVERRIDE, IMPL),                 \
- - -                                     \
```

The `ObjectIntf` prototype statement is

```
ObjectIntf    &TypeName, &Heritage [, fieldList] [, METHODS, methodList]
```

The parts of the prototype statement for `ObjectIntf` that are in square brackets is optional. A list of all fields of the object type should replace *fieldList*, with a size for each field as shown in the examples. Similarly, a list of the methods of the object type should be replaced by *methodList*. If you have a method list, the word `METHODS` must be present. Otherwise, it should never be present.

Here are some examples of its use:

```

ObjectIntf                                Tfoo, Tobject,
\
    (field1, 2),                          \
    METHODS,                              \
    (Meth1, IMPL),                        \
    (Meth2)
Meth1 ProcMethod                          Tfoo
---
EndMethod

```

The following code creates an assembly-language interface for `TObject`:

```

ObjectIntf                                TObject,, \
    METHODS,                              \
    (Free),                               \
    (ShallowFree),                       \
    (Clone),                             \
    (ShallowClone)

```

---

## ObjectWith and EndObjectWith

The `ObjectWith` and `EndObjectWith` macros allow you to specify a field of an object without having to qualify it with the object reference.

As noted earlier, you must normally qualify the name of an inherited field with the name of the object type that defined it. These macros allow you to avoid that. `ObjectWith` inserts a series of MPW assembly language `WITH` directives, one for each ancestor class, so you can specify fields of any ancestor object type without qualifying it with the ancestor object type name. `ObjectWith` can be nested. The most recent invocation has precedence when there are field-name conflicts. You end an `ObjectWith` block with an `EndObjectWith`.

These are the `ObjectWith` and `EndObjectWith` prototype statements:

```

ObjectWith      &TypeName
EndObjectWith

```

Here are two examples:

```
ObjectWith      Shape
MOVE.W          color(A1),D0
EndObjectWith

ObjectWith      Arc
MOVE.W          startAngle(A1),-(SP)
PEA             boundRect(A1)
EndObjectWith
```

---

## ProcMethodOf, FuncMethodOf, and EndMethod

The `ProcMethodOf`, `FuncMethodOf`, and `EndMethod` macros are used to bracket methods. `ProcMethodOf` and `FuncMethodOf` invoke the `PROC` and `FUNC` directives. They provide the implicit parameter `SELF`, which is a reference to the object used to call the method. They also invoke the `ObjectWith` macro with the specified type name so that fields of `SELF` can be accessed without type-name qualification. `EndMethod` invokes the `EndObjectWith` macro and the `ENDPROC` directive.

As with any assembly-language routine, before ending the routine (that is, before the `EndMethod` macro), you must remove the parameters from the stack. In a method, you should be careful to remove the implicit parameter `SELF`. (See the sample program at the end of this appendix for examples of this.)

The `ProcMethodOf`, `FuncMethodOf`, and `EndMethod` prototype statements are as follows:

```
&ProcName ProcMethodOf      &TypeName
&ProcName FuncMethodOf      &TypeName
---
EndMethod
```

Here is an example of its use:

```
Draw      ProcMethodOf      Shape
---
EndMethod

GetArea   FuncMethodOf      Arc
---
EndMethod
```

---

## MethCall

The `MethCall` macro is used to invoke methods. If the second parameter is omitted, the current method's object type is assumed. The `MethCall` macro generates a JSR to the proper method.

The `MethCall` prototype statement is

```
MethCall      &ProcName, &TypeName
```

Here is an example of its use:

```
MOVE.L      A2, -(SP)
MethCall    Draw
MOVE.W      D0, -(SP)
MOVE.W      D1, -(SP)
MOVE.L      aShape(A6), -(SP)
MethCall    MoveBy, Shape
```

---

## Inherited

The `Inherited` macro calls the named method in the closest ancestor object type that implemented the method.

The `Inherited` prototype statement is

```
Inherited    &ProcName
```

This is an example of its use:

```
MoveSelf.   -(SP)
Inherited   Draw
```

---

## NewObject

The `NewObject` macro is used to create a new object. It is equivalent to the Pascal procedure `New` used with an object reference.

`NewObject` generates a JSR to `*_OBNEW` after pushing the appropriate parameters onto the stack. If `&Size` is omitted (and it usually is), the instance size for the given object type is used. The `&Loc` parameter must be a memory reference.

The `NewObject` prototype statement is

```
NewObject      &Loc, &TypeName, &Size
```

Here are examples of its use:

```
NewObject      -4(A6), Arc  
NewObject      gArray, DynArray, 200
```

---

## MoveSelf

The `MoveSelf` macro is a convenience macro. It executes the following statement:

```
MOVE.L        8(A6), &Dest
```

`MoveSelf` assumes that the method began with a `LINK A6, #nnnn`. (`8(A6)` is the location of `SELF` when in a method.)

The `MoveSelf` prototype statement is

```
MoveSelf      &Dest
```

These are some examples of its use:

```
MoveSelf      A4  
MoveSelf      -(SP)  
MoveSelf      aSquare(A6)
```



## Appendix I Pascal and C Calling Conventions

THIS APPENDIX DESCRIBES THE CONVENTIONS USED in Pascal and C to pass parameters, return function results, and save and restore register contents during procedure and function calls. ■

### *Contents*

Pascal calling conventions	273
Parameters	273
Real-type parameters	274
Structured-type parameters	275
Function results	275
Register conventions	277
C calling conventions	277
Parameters	278
Function results	278
Register conventions	278





---

## Pascal calling conventions

This section covers parameter passing, function returns, and register conventions in Pascal.

---

### Parameters

Pascal parameters are evaluated from left to right and are pushed onto the stack in that order as they are evaluated. The called procedure is responsible for removing the parameters from the stack. All VAR parameters are passed as pointers to the actual storage location. In cases of byte-wide types, VAR parameters may have odd absolute values.

Non-VAR parameters are passed in different ways, depending on the type of the parameter. Values of type `boolean`, elements of an enumerated type with fewer than 128 elements, and subranges within the range `-128..127` are passed as signed byte values. (They are pushed as bytes; the 68000 allocates two bytes for each byte on the stack.) The called procedure expects `boolean` parameters to be in the range `0..1`. Values of types `integer`, `char`, and all other enumerations and subranges are passed as signed word values. In Pascal, values of type `char` are expected to be in the range `0..255`; the upper half of this range is used for special characters. Pointers and `longint` values are passed as signed 32-bit values.

Table I-1 summarizes the Pascal parameter passing conventions.

■ Table I-1 Parameter passing conventions

Parameter type	Pascal caller Action	Pascal receiver Action
<code>boolean</code>	Pushes byte: range <code>0..1</code>	Accesses byte: range <code>0..1</code>
enumeration: range <code>0..127</code>	Pushes byte: range <code>0..127</code>	Accesses byte: range <code>0..127</code>
enumeration: range <code>0..32767</code>	Pushes word: range <code>0..32767</code>	Accesses word: range <code>0..32767</code>
<code>char</code>	Pushes word: range <code>0..255</code>	Accesses word: range <code>0..255</code>
subrange: range <code>-128..127</code>	Pushes byte: range <code>-128..127</code>	Accesses byte: range <code>-128..127</code>

(continued)

■ Table I-1(continued) Parameter passing conventions

Parameter type	Pascal caller Action	Pascal receiver Action
subrange: range -32767..32767	Pushes word: range -32767..32767	Accesses word: range -32767..32767
integer	Pushes word: range -32767..32767	Accesses word: range -32767..32767
longint	Pushes long	Accesses signed long value
pointer	Pushes long	Accesses long
real	Converts to extended; pushes address of extended	Converts extended on stack to local real; accesses local value
double	Converts to extended; pushes address of extended	Converts extended on stack to local double; accesses local value
comp	Converts to extended; pushes address of extended	Converts extended on stack to local comp; accesses local value
extended	Pushes address of extended	Copies extended to local extended; accesses local value
ARRAY, RECORD, STRING ≤ four bytes	Pushes value (word or long)	Accesses value (word or long)
ARRAY, RECORD, STRING > four bytes	Pushes address of value	Copies value to local; accesses local
SET	Pushes set value rounded to whole number of words	Accesses value on stack (Note: Use word or long for those sizes; accesses low-order half of word for byte-size set.)

### Real-type parameters

Values of types `real`, `double`, `comp`, and `extended` are passed as pointers to `extended` values. The Compiler does this in a reentrant way by allocating a temporary location in the caller's activation record, converting the parameter value to an `extended` value in this location, and passing a pointer to this location. The called procedure then allocates a local location of the declared type and converts the `extended` value, using the pointer, into the location and type.

## Structured-type parameters

Arrays, strings, and records whose size is less than or equal to 4 bytes are passed by pushing their value (either a word or a long word) onto the stack. Larger arrays, strings, and records (as well as `extended` values, as mentioned earlier in this appendix) are passed as a pointer to the value; for reentry purposes, the Compiler emits code in the called procedure to copy the value to a local storage location.

Sets are passed by rounding the set size up to the next whole word, if necessary, then pushing the set value so that the lowest-order word is pushed last. In the case of a byte-width set, the called procedure will only access the low-order half of the word pushed.

---

## Function results

Function results are returned by value or by address on the stack. Space for the function result is allocated by the caller before the parameters are pushed. The caller is responsible for removing the result from the stack after the call.

For types `boolean`, `char`, `integer`, and enumerated and subrange types, the caller allocates a word on the stack to make space for the function result. Values of type `boolean`, enumerated types with fewer than 128 elements, and subranges within the range `-128..127` are returned as signed byte values. The value goes in the low-address byte, which is the most significant byte of the word. The calling procedure expects `boolean` results to be in the range `0..1`.

`Integer` and `char` values and all enumerated and subrange types not covered above are returned as signed word values. Pascal `char` values are expected to be in the range `0..255`; the upper half of this range is used for special characters.

For `pointer`, `longint`, and the `real` types, the caller allocates a long word on the stack to make space for the function result. Pointers and `longint` values are returned as signed 32-bit values. Values of type `real` are returned as 32-bit real values. For `double`, `comp`, and `extended` types, and also for sets, arrays, strings, and records greater than 4 bytes in size, the caller pushes a pointer to a temporary location.

For 1-byte sets and for arrays, strings, and records whose size is 1 word, the caller allocates a word on the stack. For sets, arrays, strings, and records whose size is 2 words, the caller allocates a long word on the stack. 1-byte sets are returned as a byte value. Sets, arrays, strings, and records whose sizes are 1 or 2 words are returned as either a word or a long word.

Pascal function-result passing conventions are summarized in Table I-2.

■ Table I-2 Function-result passing conventions

Parameter type	Pascal caller Action	Pascal receiver Action	After the call
boolean	Allocates word	Returns byte value: range 0..1	Pops byte
enumeration: range 0..127	Allocates word	Returns byte value: range 0..127	Pops byte
enumeration: range 0..32767	Allocates word	Returns word value: range 0..32767	Pops word
char	Allocates word	Returns word value: range 0..255	Pops word
subrange: range -128..127	Allocates word	Returns byte value: range -128..127	Pops byte
subrange: range -32768..32767	Allocates word	Returns word value: range -32768..32767	Pops word
integer	Allocates word	Returns word value: range -32768..32767	Pops word
longint	Allocates long word	Returns long word value: range—signed 32 bits	Pops long word
real	Allocates long word	Returns real value	Pops real value
double	Pushes address of double temporary	Puts double result in temporary	Pops temporary address, accesses temporary value
comp	Pushes address of comp temporary	Puts double result in temporary	Pops temporary address, accesses temporary value
extended	Pushes address of extended temporary	Puts extended result in temporary	Pops temporary address, accesses temporary value
ARRAY, STRING, RECORD ≤ four bytes	Allocates word or long word	Returns word or long word	Pops word or long word

■ **Table I-2 (continued)**      Function-result passing conventions

Parameter type	Pascal caller Action	Pascal receiver Action	After the call
ARRAY, STRING, RECORD > four bytes	Pushes address of temporary	Puts result in temporary	Pops temporary address, accesses temporary value
SET: one byte	Allocates word	Returns byte value of result	Pops byte
SET: one word	Allocates word	Returns word value of result	Pops word
SET: two words	Allocates long word	Returns long word value of result	Pops long word
SET > two words	Pushes address of temporary	Puts result in temporary	Pops temporary address, accesses temporary value

- ◆ *Note:* Pascal does not assume any initial value for memory space allocated to a function result unless it is a pointer to a type that occupies more than 4 bytes of memory.

---

### Register conventions

Registers D0, D1, D2, A0, and A1 are considered scratch registers and are not preserved across procedure calls. All other registers are preserved by the called routine. Register A5 is the global frame pointer, register A6 the local frame pointer, and register A7 the stack pointer.

---

### C calling conventions

This section covers the treatment of parameters, function results, and registers in C.

---

## Parameters

Parameters to C functions are evaluated from right to left and are pushed onto the stack in the order they are evaluated. Characters, integers, and enumerated types are passed as sign-extended 32-bit values. Pointers and arrays are passed as 32-bit addresses. Types `float`, `double`, `comp`, and `extended` are passed as extended 80-bit values. Structures are also passed on the stack. Their value is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the highest memory address. The caller removes the parameters from the stack.

---

## Function results

Characters, integers, enumerated types, and pointers are returned as sign-extended 32-bit values in register D0. Types `float`, `double`, `comp`, and `extended` are returned as extended values in registers D0, D1, and A0. The low-order 16 bits of D0 contain the sign and exponent bits; register D1 contains the high-order 32 bits of the significand; register A0 contains the low-order 32 bits of the significand. Structured values are returned as a 32-bit pointers in register D0. The pointer contains the address of a static variable into which the result is copied before returning. This implementation of structured function results is not reentrant.

---

## Register conventions

Registers D0, D1, A0, and A1 are scratch registers that are not preserved by C functions. All other registers are preserved. Register A5 is the global frame pointer, register A6 is the local frame pointer, and register A7 is the stack pointer. Local stack frames are not necessarily created for simple functions.

## Appendix J Structured Assembly Macros

HIGHER-LEVEL LANGUAGES, SUCH AS PASCAL AND C, provide the programmer with statements that represent basic structured programming operations. There are statements for looping (FOR, WHILE, REPEAT), conditionals (IF-THEN-ELSE, SWITCH, CASE), procedure declarations (PROCEDURE, FUNCTION), and procedure invocation. The compilers accept the high-level statements and generate the proper code for the programmer. Assembler programmers must code all these constructs explicitly. The structured assembly macros are used as high-level statements that serve the same purpose as their high-level language counterparts, and thus relieve the programmer of such explicit coding. ■

### *Contents*

Structured macro statements	281
Expressions	281
Flow-control macros	283
The If statement	283
The Switch statement	285
The Repeat statement	287
The While statement	287
The For statement	288
The Leave statement	290
The Cycle statement	291
The GoTo statement	292

Program structure macros	292
Sample code generation from program structure macros	294
Procedure and function header	295
Local variable declaration	298
Procedure or function start	299
Procedure or function secondary entry point	300
Procedure or function exit	301
Procedure, function, or trap invocation	303
Considerations for use	306
Why you should or should not use the structured assembly macros	307
Rules for using structured assembly macros	308
Syntax summary	309
Expressions	309
Flow-control macros	310
Program structure macros	311



---

## Structured macro statements

The macros are divided into two categories: *program structure macros* and *flow-control macros*.

The program structure macros are as follows:

PROCEDURE	Define a procedure declaration
FUNCTION	Define a function declaration
VAR	Declare procedure or function local variables
BEGIN	Define a procedure or function primary entry point
ENTER	Define a procedure or function secondary entry point
RETURN	Exit from a procedure or function
CALL	Invoke a procedure, function, or trap

The flow-control macros are as follows:

IF#, ELSEIF#, ELSE#, ENDF#	Multiway decision
SWITCH#, CASE#, DEFAULT#, ENDS#	Multiway decision
REPEAT#, UNTIL#	Loop control
WHILE#, ENDW#	Loop control
FOR#, ENDF#	Loop control
LEAVE#	Loop and switch terminator
CYCLE#	Loop iterator
GOTO#	Transfer of control

With the exception of assignment statements and full arithmetic expressions, these macros provide all the constructs found in high-level languages such as Pascal and C. Because of the restrictions imposed on the kinds of conditional expressions allowed in these macros (discussed later), the code generated for these statements is as efficient as that any programmer can generate "by hand."

---

## Expressions

Expressions are used as operands to many of the flow-control macros. Such expression operands are used for testing conditions. The following syntax describes what is allowed for flow-control macro expressions:

```

expr ::= s-expr | s-expr op s-expr
s-expr ::= cc | ea cc [.sz] ea
op ::= AND | OR
cc ::= EQ | NE | LE | LT | GE | GT | MI | PL | HI | LS |
        LO | CC | CS | NZ | HS | VC | VS
sz ::= B | W | L

```

An expression, *expr*, consists of either one simple expression, *s-expr*, or two simple expressions combined by AND or OR. Simple expressions, in turn, either test a condition code, *cc*, or set and test a condition code by comparing two operands, each of which is specified by an effective address, *ea*. The comparisons cause the macros to generate compare instructions. To indicate the size of the comparison, an optional size, *sz*, may be specified along with the condition code. Word comparisons are the default.

Note that the effective address and size for the comparisons must be valid for the form of compare instruction generated. For example, byte size cannot be specified for comparisons involving address registers, and both effective addresses must specify a post increment if a memory-to-memory comparison is to be done. However, the order of the comparison will be reversed by the macros if a legal comparison cannot be generated as specified, but it would be legal if the comparison were reversed. The condition being tested would be similarly reversed.

The effective address operands may be arbitrarily complex effective addresses, which may contain full Assembler expressions. AND and OR operators are also legal operators when used in Assembler expressions. When using these operators in both Assembler and macro expressions, the AND and OR macro expression operators will be recognized only if not nested in paired parentheses. The use of the condition code symbols, *cc*, is somewhat more restrictive. They should only be used as shown in the syntax and never for identifiers used in effective addresses anywhere else.

Macro expressions combined by AND and OR will generate the comparisons determined by the two simple expressions in the order given. However, for OR operations, the second operand will not be executed if the first operand is true. Similarly, for AND operations, the second operand will not be executed if the first operand is false. In these cases, the resulting condition code would only reflect the result of the first simple expression.

```

NE          #'$' LE.B D5          MI OR VS
D0 NE.W D1  10(A5) NE.L D3        D0 EQ.B #$13 OR D0 EQ.B #$12
D0 NE.B D1  D3 NE.B 10(A5)        (A2)+ EQ #' ' OR (A2) EQ #$13
#5 GT D0    (A2)+ EQ.B (A3)+      (X+10) (A2,D2.W) EQ.B D3 OR NE
D0 EQ.B #'*' ([10,A5],D2) NE D3  ([X,A5,D3],(Y).L) NE.L D6

```

---

## Flow-control macros

The flow-control macros provide a full set of the standard conditional and loop control statements found in most higher-level languages. The flow-control macros are as follows:

IF#, ELSEIF#, ELSE#, ENDIF#	Multiway decision
SWITCH#, CASE#, DEFAULT#, ENDS#	Multiway decision
REPEAT#, UNTIL#	Loop control
WHILE#, ENDW#	Loop control
FOR#, ENDF#	Loop control
CYCLE#	Loop iterator
LEAVE#	Loop and switch terminator
GOTO#	Transfer of control

---

### The If statement

```
IF# expr1 THEN[.ext]
    statements1
```

```
[ ELSEIF#[.ext] expr2 THEN[.ext2] ]
    statements2
```

```
[ ELSE#[.ext] ]
    statements3
```

```
ENDIF#
```

If *expr*<sub>1</sub> is true, execute only the statement list, *statements*<sub>1</sub>. If *expr*<sub>1</sub> is false, execute the statement list for the first true ELSEIF# clause if present. If all the expressions are false, then execute the statement list, *statements*<sub>3</sub>, if the ELSE# clause is present; otherwise skip to the first statement following ENDIF#.

There may be any number of `ELSEIF#` clauses, but only one `ELSE#` clause, between the `IF#` and `ENDIF#` pair. If the `ELSE#` clause is present, it must follow all the `ELSEIF#` clauses. If statements may be nested to an implementation-defined limit (see "Considerations for Use").

The optional extension attributes, *ext's*, are the letters S or B, W, or L, and control the size of the branch instructions generated during compilation of the If statement.

- The extension on the `THEN` portion of the `IF#` determines the size of the branches to the next `ELSEIF#`, `ELSE#`, or `ENDIF#`, whichever comes first.
- The `ELSEIF#` extension, *ext<sub>1</sub>*, determines the size of the branch to the `ENDIF#`.
- The extension, *ext<sub>2</sub>*, on the `THEN` portion of the `ELSEIF#` determines the size of the branches to the next `ELSEIF#`, `ELSE#`, or `ENDIF#`, whichever comes first.
- The `ELSE#` extension determines the size of the branch to `ENDIF#`.

The default for all the branch extensions is for word (w) branches. Here is an example:

```
IF# D0 EQ.B #$20 OR D0 EQ.B #$09 THEN.S
    JSR  SkipBlanks
ELSEIF#.S D0 GE.B #'A' AND D0 LE.B #'Z' THEN.S
    JSR  Identifier
ELSEIF#.S D0 GE.B #'a' AND D0 LE.B #'z' THEN.S
    JSR  Identifier
ELSEIF#.S D0 GE.B #'0' AND D0 LE.B #'9' THEN.S
    JSR  Number
ELSE#.S
    JSR  Special
ENDIF#
```

This example checks for characters in `D0` and calls an appropriate processing routine as a function of the character. For blanks (`$20`) and tabs (`$09`), `SkipBlanks` is called. For uppercase and lowercase letters, `Identifier` is called. For digits, `Number` is called, and if the character is anything else, `Special` is called. Note that all the extensions are `s` to cause all short branches to be generated.

---

## The Switch statement

```
SWITCH# [.sz] selector [, Dreg=Dn] [, JmpTbl={N|Y|ext}], [ChkRng={N|Y}]
CASE# [.ext] ae1 [...ae2], ...
    statements1
[ DEFAULT#
  statements2 ]
```

ENDS#

The SWITCH statement is a multiway branch based on the value of *selector*. Each CASE# absolute expression, *ae*<sub>1</sub>, specifies a list or range of constants representing a value of *selector*. If *selector* equals one of the constants, then the statements following that CASE# are executed. If *selector* is not one of the values, and the DEFAULT# clause is present, the statements following the DEFAULT# are executed. If *selector* is not one of the values, and there is no DEFAULT# clause, then none of the statements of the SWITCH statement are executed and control passes to the first statement following ENDS#. After control passes to one of the CASE or DEFAULT statements, execution continues through successive statements until the end of the SWITCH statement, ENDS#, is reached, or control is transferred out of the SWITCH structure (for example, using a LEAVE# macro).

There may be any number of CASE# clauses, but only one DEFAULT# clause, between the .i. SWITCH# and ENDS# pair. There is no restriction on the placement of the DEFAULT# clause within the SWITCH statement. SWITCH statements may be nested to an implementation-defined limit (see "Considerations for Use").

The code for SWITCH statements is generated to do either repeated subtractions from the *selector* value to determine the case, or to use a relative address jump table indexed by the *selector* value. The default is to use repeated subtractions unless either JmpTbl=Y or JmpTbl=ext is specified. Using the jump table, you have the option of validating the *selector* value to make sure it is in the proper index range of the jump table, and using the default (or skipping the SWITCH statement) if it is out of range. This is specified by ChkRng=Y. The default is to not validate the *selector* value when a jump table is used (and therefore no DEFAULT# clause is possible).

The choice of whether to use the repeated subtraction technique or a jump table is up to you. It generally depends on the number of cases and the distribution of case values. If you choose the repeated subtraction technique, then you may specify a branch size, *s* or *B*, *w*, or *L*, as an extension attribute on the *CASE#* statements, to indicate the size of the branches to the next *CASE#* or *DEFAULT#* clause. If you choose the jump table technique, then the *CASE#* extensions are ignored, but you may specify the size of the branch required to branch from *SWITCH#* to *ENDS#*, where the actual indexed jump table code is generated. This is specified by an explicit extension value for the *SWITCH# JmpTbl* parameter; that is, *JmpTbl=S|B|W|L*. *JmpTbl=Y* is equivalent to *JmpTbl=W*.

No matter which technique is used, *SWITCH* statements require a work register in which to do the subtractions or to convert to a table index. This is specified by the *SWITCH# DREG* parameter, or *D0* is used by default. If you explicitly specify the *DREG* parameter, then the specified *D*-register will be loaded from *selector*. Whenever *selector* is placed in the work register, you may indicate the size attribute of the move instruction, *B* or *w*, to do the load by specifying an attribute, *sz*, on the *SWITCH#* macro call. The default is to assume a word move. The work register is always used as a word (note that *selector* will not be moved to the work register if *D0* is specified as *selector*, but in that case it is assumed that *D0* already contains a word value).

```

SWITCH# D0
CASE#.S $20, $09
    JSR  SkipBlanks
LEAVE#.S

CASE#.S 'A'..'Z', 'a'..'z'
    JSR  Identifier
LEAVE#.S

CASE#.S '0'..'9'
    JSR  Number
LEAVE#.S

DEFAULT#
    JSR  Special
ENDS#

```

This example checks for characters in *D0* (previously loaded as a word) and calls an appropriate processing routine as a function of the character. It is the same example shown for *IF* statements, but here rewritten using a *CASE* statement. Repeated subtractions and all short branches are used. The *LEAVE* statements are used to terminate each case (see "The Leave Statement" later in this appendix).

---

## The Repeat statement

```
REPEAT#  
    statements  
UNTIL#[ext] (expr | FALSE)
```

The statements between REPEAT# and UNTIL# are executed at least once and then repeatedly until *expr* is true. FALSE may be specified in place of *expr* to generate an infinite loop. The size of the branch instructions, S or B, W, or L, generated by UNTIL# to loop back to REPEAT# may be specified by the extension attribute, *ext*.

REPEAT statements may be nested to an implementation-defined limit (see "Considerations for Use" later in this appendix).

```
REPEAT#  
    PEA    filterProc  
    PEA    itemHit(A6)  
    ModalDialog  
UNTIL#.S itemHit(A6) EQ #OK
```

This example calls `ModalDialog` until the OK button is clicked. A short branch is used to branch back to the top of the loop.

---

## The While statement

```
WHILE# (expr | TRUE) DO[ext]  
    statements  
ENDW#
```

The statements between WHILE# and ENDW# are executed repeatedly only if *expr* is true. If *expr* is false, control passes to the first statement following ENDW#. True may be specified in place of *expr* to generate an infinite loop. The size of the branch, S or B, W, or L, generated by the WHILE# to ENDW# structure is specified by the extension attribute following the DO keyword.

While statements may be nested to an implementation-defined limit (see "Considerations for Use" later in this appendix). Here is an example:

```
SkipBlanks    WHILE# D0 EQ.B #$20 OR D0 EQ.B #$09 DO.S  
              JSR    NextChar  
              ENDW#
```

This example illustrates a possible blank and tab skipping routine that might be called by the IF# or CASE# example. As long as D0 contains a blank (\$20) or tab (\$09), a NextChar routine is called, which loads D0 with the next input character. As soon as D0 does not contain a blank or tab, a short branch (DO . S) is taken to the statement following ENDW#.

---

## The For statement

```
FOR# ctl-var [= [sz] initial] [DOWN]TO final [BY increment] [UNTIL expr] DO [ ext ] \  
      [,DREG=Dn] [,Opt={Y | N}] [,CLR={Y|N}]  
      statements  
ENDF#
```

The statements between FOR# and ENDF# are executed repeatedly while the control variable, *ctl-var*, is assigned a progression of values starting with the *initial* value. If the *initial* value is greater than (TO) or less than (DOWNTO) the final value on entry to the FOR statement, control passes to the first statement following ENDF#. Otherwise, the control variable is incremented (TO) or decremented (DOWNTO) by *increment* (default 1). Incrementing continues on each repetition of the loop until the value of the control variable is greater than (TO) or less than (DOWNTO) the final value, or until the value of *expr* is true. The optional UNTIL *expr* clause is similar in function to a REPEAT statement's UNTIL clause. It is only processed at the end of the loop and thus does not control whether the loop will be entered the first time.

For statements may be nested to an implementation-defined limit (see "Considerations for Use" later in this appendix).

The variables *ctl-var*, *initial*, *final*, and *increment* all represent effective addresses. The control variable must be an alterable effective address mode. All effective address modes are allowed for the initial, final, and increment specifications. The size of the control variable, B, W, or L, is determined by the size attribute, *sz*, following the equal sign assigning the initial value to the control variable. The default is to assume a word size control variable. Note that the initial value may be omitted, which implies a word size control variable that already has its initial value.

The size of the branch statements, S or B, W, or L, generated by the FOR statement is determined by the extension on DO. The default is to assume word size branches.



For statements always require the use of one work D-register. If the control variable already specifies a D-register, that register is used and may be referenced by the loop statements as usual. If the control variable does not specify a D-register, then the D-register specified by the `DREG` parameter is used, or `D0` if `DREG` is not specified. In that case the control variable is still maintained and can be referenced by the loop statements. Although the work register also contains the current control variable value, it should not be considered safe across the loop.

As you can see, the most efficient `FOR` loop code is generated when the control variable is a D-register. If it isn't, additional code is generated to copy the control variable to and from a work register while incrementing or decrementing it. The incrementing and decrementing itself is done by explicit `ADD` or `SUB` instructions, and the end of loop condition is tested with a `CMP` instruction.

For a restricted class of `FOR` loop operands the generated code can be further optimized to use a `DBCC` instruction. The following four `FOR` statements will generate `DBCC` instructions:

```
FOR# Dn [= [.sz] initial] DOWNTO #0 [BY #1] [UNTIL expr] DO
FOR# Dn [= [.sz] initial] TO #0 BY #-1 [UNTIL expr] DO
FOR# Dn [= [.sz] initial] DOWNTO #1 [BY #1] DO
FOR# Dn [= [.sz] initial] TO #1 BY #-1 DO
```

The first two `FOR` statements allow an `UNTIL expr` clause. If *expr* is just a condition code, the `DBCC` instruction generated will be a function of that condition code. If *expr* is anything more than a condition code, a `DBF` is generated, but additional code will be generated to implement an `IF` statement to test the terminating *expr*. If the `UNTIL expr` clause is omitted, or one of the last two `FOR` statement classes is specified, a `DBF` is generated.

In all four special cases, the size, *sz*, for the control variable must be byte or word. Since `DBCC` instructions require a word size register value, if the size is specified as byte, the control variable must be cleared (`CLR.W`) prior to setting its initial value. This clearing process may be suppressed by specifying the `CLR=N FOR#` parameter, or loading the control variable (as a word) prior to `FOR#`, and not specifying an initial value. If you don't want any of the optimizations, and would rather have an explicit increment or decrement along with the accompanying `CMP` instruction, you may suppress the `DBCC` optimization by specifying the `FOR# OPT=N` parameter.

```
FOR# D0=0 TO #4*(N-1) BY #4 DO.S
IF# 0(A0,D0.W) GT D1 Then.S
MOVE.W 0(A0,D0.W),D1
ENDIF#
ENDF#
```

This example scans N word values pointed to by A0 and returns the maximum value in D1 (assumed initialized). Short branches are used. Since the FOR statement does not fall into one of the four possible DBCC optimization classes, an explicit ADD and CMP will be generated. Here is an example:

```
FOR# D0=#63 DOWNT0 #0 DO.S
    MOVE.L    (A0)+, (A1)+
ENDF#
```

This example copies 256 bytes from the area pointed to by A0 to the area pointed to by A1. A DBF loop is generated to loop 64 times (0 to 63) to move 4 bytes at a time.

---

## The Leave statement

LEAVE#[*ext*] [*label*] [IF[#]*expr*]

The LEAVE statement causes execution of the smallest enclosing FOR, WHILE, REPEAT, or SWITCH statement to be terminated. A label may be specified to indicate an enclosing FOR, WHILE, REPEAT, or SWITCH statement that is at a higher nesting level than the one containing this LEAVE statement. All loops and switches up to and including the one associated with the label are terminated. The LEAVE statement may be made conditional by specifying the IF clause.

LEAVE statements cause a generation of a branch to the first statement following the loop or switch to be terminated. The size of the branch, s or B, w, or L, may be specified with the extension attribute, *ext*.

```
FOR# D0=0 TO #4*(N-1) BY #4 DO.S
    LEAVE#.S IF TABLE(A5,D0.W) EQ D1
ENDW#
```

This example searches a table of values for the value in D1 and stops when all the elements of the table are searched or the value is found, whichever occurs first. Here is an example:

```
Outer FOR# D0=#1 TO #N DO.S
Inner  FOR# D1=#1 TO #M DO.S
      - - -
      LEAVE# Outer IF D3 EQ D4
      - - -
      ENDF#
    ENDF#
```

In this example, both loops are terminated when D3 equals D4. The explicit label, *outer*, indicates which loop is to be terminated. If the label were omitted, only the inner loop would be terminated.

LEAVE statements used in switches are illustrated in the example for SWITCH statements (see "The Switch Statement," given earlier in this appendix).

---

## The Cycle statement

CYCLE#[*ext*] [*label*] [IF[#]*expr*]

The CYCLE statement causes the next iteration of the smallest enclosing FOR, WHILE, or REPEAT statement to be executed. A label may be specified to indicate an enclosing FOR, WHILE, or REPEAT statement that is at a higher nesting level than the one containing this CYCLE statement. All loops enclosing this CYCLE statement are terminated and the one associated with the label is iterated. The CYCLE statement may be made conditional by specifying the IF clause.

CYCLE statements cause a generation of a branch to the loop continuation part of the associated loop statement. The size of the branch, S or B, W, or L, may be specified with the extension attribute, *ext*.

Note that CYCLE statements do not apply to SWITCH statements. A CYCLE statement inside a SWITCH statement nested in a loop causes the loop to be iterated. Here is an example:

```
Outer WHILE# D0 NE.L 0 DO.S
Inner WHILE# D1 NE.L 0 DO.S
- - -
IF# D3 EQ D4 THEN.S
CYCLE# Outer
ELSEIF#.S D4 EQ.L #0 THEN.S
CYCLE# Inner
ENDIF#
- - -
ENDW#
ENDW#
```

In the example given here, when D3 equals D4, the inner loop is terminated and the outer loop is iterated. If D3 does not equal D4, but D4 is zero, then the inner loop is iterated. The explicit labels indicate which loop is to be terminated. In the case of CYCLE# Inner, you could have omitted the label reference.

---

## The GoTo statement

`GOTO#[.ext] [IF[#]expr THEN[.ext]]label`

The `GOTO` statement is included only for the sake of completeness. A `BRA` instruction is generated for the `GOTO` statement. The branch is made conditional by specifying the `IF` clause. The size of the branch, `S` or `B`, `W`, or `L`, is specified by the extension, *ext*. It may be specified either on the `GOTO#` or `THEN`. The rightmost extension is the one used.

---

## Program structure macros

The program structure macros are used to define procedures and functions, to define local variables belonging to them, and to call them. The macros are as follows:

<code>PROCEDURE</code>	Define a procedure declaration header
<code>FUNCTION</code>	Define a function declaration header
<code>VAR</code>	Declare procedure or function local variables
<code>BEGIN</code>	Define a procedure or function starting point
<code>ENTER</code>	Define a procedure or function secondary entry point
<code>RETURN</code>	Exit from a procedure or function
<code>CALL</code>	Invoke a procedure, function, or trap

These macros make it easier to define and call modules. They take much of the burden off you as a programmer, who would otherwise have to explicitly define formal parameters and local variables using either equates or template mappings. The syntax is also more readable, and taken together with the `CALL` macro, tends to compress an Assembler source file so that it takes fewer source lines to define and call a procedure.

With the exception of the `CALL` macro, all the other program structure macros are used to define a procedure or function code module. They have a fixed relationship with one another. That relationship is governed by the standard rules imposed by the Assembler on defining code modules.

To define a code module you first must have a `PROC`, `FUNC`, or `MAIN` directive to delimit the scope of the code module and the local variables declared inside it. Inside the module you may define a template to map over the formal parameters and local variables to be placed in the module's stack frame. The entry point of the code module possibly does a `LINK` to set up the stack frame, and saves any nonvolatile registers. Finally comes the code for the module followed by a restore of the saved registers, an `UNLK` to delete the stack frame, and a return to the caller. Thus the macros, which perform these functions, are specified in the order illustrated by the following example:

```

Export FUNCTION EXAMPLE(Arg1:L, Arg2, Arg3:B):L      ;Function hdr, formals, result
VAR          Local1:L, Local2:LEN                  ;Local variables
VAR          Local3:B[256], Local4:T1;
BEGIN       SAVE=D3-D5/A2-A4, WITH=(Globals,T2)    ;Entry point, reg save, LINK
- - -                                             ;code for module...
RETURN     DO                                      ;Exit with DO as result
ENDP                                              ;End module

```

The PROCEDURE or FUNCTION macro is responsible for setting up the PROC or FUNC directive and declaring the module's scope. The example just given shows an exported function with a result. ENTRY procedures or functions are also possible, as well as embedded (local) procedures. (An embedded procedure is one that is local to another procedure.)

In addition to declaring the module's scope, the header also declares any formal parameters. These are translated into a template mapping over a stack frame. The exact syntax for declaring formal parameters is discussed in "Procedure and Function Header," given later in this appendix. In general, the syntax allows you to declare the name, size, and a repetition count of each formal. Sizes can be specified as bytes, words, longs, absolute expressions, or as template names. The latter case also defines the type of the formal parameter.

If local variables are to be declared, as they are in this example, they are added to the stack frame definition started by the procedure header by using the VAR macro. The functionality for declaring local variables is much the same as that for formal parameters. As many VAR macros as desired may be specified to declare all the local variables needed by the procedure. They may be declared one variable per call or many variables per call.

All the VAR local variable declarations are placed between the procedure header and the BEGIN macro. The BEGIN macro marks the start of the procedure code body. The stack frame template definition is closed and LINK and MOVEM instructions are generated as required. With BEGIN you specify any registers to be saved and additional templates or data modules to be converted with a WITH directive. The stack frame is always covered by a WITH directive, and the stack frame pointer, A6 or A7, is equated to the local name FP. Using FP you can access the local variables and formals on the stack via the stack frame template.

At the end of the procedure you use the RETURN macro to exit. The RETURN macro generates a MOVEM statement to restore any registers saved by the BEGIN macro. The UNLK is also done and code is generated to pop the arguments off the stack and to return to the caller. Depending on the PROCEDURE macro parameters you can also generate the ASCII module name following the exit so that the procedures can be debugged using MacsBug.

As you can see, the program structure macros do a lot of work for you. All the lines except the ones marked as comments with an asterisk are generated by the macros. The following sections will discuss each macro in detail.

---

### Sample code generation from program structure macros

```

* Export FUNCTION      EXAMPLE (Arg1:L, Arg2, Arg3:B):L
                        ;Function hdr, formals, result

Example  PROC          Export
SF#xxxx RECORD        {FramePtr},Decr
Example  DS.L          0
Arg1     DS.L          1
Arg2     DS.W          1
Arg3     DS.W          1
RetAddr  DS.L          1
*        VAR          Local1:L, Local2:Len      ;Local variables
LinkA6   DS.L          1
FramePtr EQU          *
Local1   DS.L          1
Local2   DS.W          LEN
*        VAR          Local3:B[256], Local4:T1;
Local3   DS.B          256
Local4   DS.W          T1
*        BEGIN        SAVE=D3-D5/A2-A4,WITH=(Globals,T2)
                        ;Entry point, reg save, LINK
LocalSize DS.W         0
                        ENDR
                        WITH      Globals,T2,SF#xxxx
                        LINK      A6,#LocalSize
FP       SET          A6
                        MOVEM     D3-D5/A2-A4,-(A7)
                        - - -
*        RETURN       D0              ;Exit with D0 as result
MOVEM.L  (A7)+,D3-D5/A2-A4
UNLK     A6
MOVEA.L  (A7)+,A0
ADD.W    #8,A7          ;Optimizes to ADDQ
MOVE.L   D0,(A7)
JMP     (A0)
ENDP

```

---

## Procedure and function header

$$\left[ \begin{array}{l} \text{ENTRY} \\ \text{EXPORT} \\ \text{LOCAL} \end{array} \right] \left\{ \begin{array}{l} \text{PROCEDURE} \\ \text{FUNCTION} \end{array} \right\} [\text{module-id}] ([ [\text{formal}, \dots] ] : [\text{result}] ) [C] [,Link=(Y|DEBUG)] \backslash \\ [,Main=(N|Y)]$$

where

$$\begin{array}{ll} \text{formal} & ::= \text{id} : \text{formal-sz} [\text{count}] \\ \text{formal-sz} & ::= \text{B} | \text{W} | \text{L} | \text{S} | \text{D} | \text{X} | \text{P} | \text{ae} | \text{template-id} \\ \text{count} & ::= \text{'(ae)'} \\ \text{result} & ::= \text{B} | \text{W} | \text{L} | \text{S} | \text{D} | \text{X} | \text{P} | \text{id} \end{array}$$

The procedure and function header macros are used to do the following things:

- **Declare a new code module as a procedure or function.**

The two macro calls, `PROCEDURE` and `FUNCTION`, are interchangeable and may be used as appropriate for documentation purposes. For simplicity, both procedures and functions are referred to as procedures in the following discussion. A main program entry point may be specified using the `Main=Y` macro parameter.

- **Define the procedure's scope.**

A procedure may be declared as `ENTRY` (the default), `EXPORT`, or `LOCAL`. These words are specified in the label field of the macro call. An entry or export procedure is declared using the standard Assembler `PROC` or `FUNC` directives, with `ENTRY` or `EXPORT` as the directive's parameter and the module name, *module-id*, as the directive's label (note that the macro syntax is thus inverted from that of the corresponding Assembler directives). A local procedure does not cause generation of a `PROC` or `FUNC` directive. This form of declaration is used to define an inner procedure where only the label (*module-id*) is defined. An `ALIGN 2` directive always precedes a local procedure declaration.

- **Declare a procedure's formal parameters (if any) that are to be passed on the stack.**

As shown in the syntax description at the beginning of this section, the formal parameter list is enclosed in parentheses. The formal parameters' identifiers are defined by placing them in a stack frame template definition started by the macro (discussed later in this appendix).

Each formal parameter identifier may be followed by a size attribute. If the size is not specified, word (w) is assumed. If the size is specified it may be any of the standard Assembler size attributes, an absolute expression, or a template name. An absolute expression explicitly specifies the size of the formal. A template name defines both the size and the type of the formal (see description of template types elsewhere in this Reference).

Following the size you may specify a repeat count enclosed in required brackets. The default repeat count is 1. The repeat count indicates how repetitions of the basic size are to be allocated for the formal parameter. Thus the amount of space actually allocated for a formal is *count \* formal-sz*. Note that since you are describing the size of parameter passed on the stack, and assumed pushed on the stack using A7, all sizes are rounded up to an even value. Specifying a byte actually allocates a word, and absolute expressions are rounded up to an even value.

Normally, each formal parameter is processed left to right, causing a new entry in the stack frame template set up for this procedure. However, for C procedures, you may have the option of processing the formal parameters right to left, so that the leftmost argument is the one closest to the top of the stack. Right-to-left processing is specified by using the C parameter, as indicated in the syntax.

- **Define a function result.**

By placing *result* after the formal parameter list (if any), you indicate that a function result is to be returned (you can still use the Procedure macro even if this is a function). The *result* parameter specifies the size or an identifier. If you specify a size, then the assumed identifier is the same as the procedure's name, *module-id*. Whichever name is used, that name may be referenced as a stack frame offset to store the function result; for example, `MOVE .w DO,name(FP)`. More specifically, the name is defined as the first stack frame entry (template) field with no space allocated to it (if you specify a size it is used for commentary purposes only). In this position you could set the function result after all arguments have been popped off the stack prior to exit. See "Procedure or Function Exit," given later in this appendix, for further details on how the function result is set.

- **Force a LINK to be done at entry (by the BEGIN macro) and the corresponding UNLK to be done at exit (by the RETURN macro).**

Normally the LINK/UNLK pair is automatically generated for a procedure when you specify local variables (with the VAR macro), or have saved registers (specified by the BEGIN macro) and a function *result* or formal parameters. However, you may force generation of the LINK/UNLK pair by specifying the LINK=Y or LINK=DEBUG parameter on the PROCEDURE or FUNCTION macro call. The LINK=DEBUG is the same as LINK=Y, but in addition causes the procedure's *module-id* (up to its first eight characters) to be generated following the exit code. This allows you to use MacsBug, which requires the LINK/UNLK pair and name so it can identify the module in memory.



It is possible that you might want to turn on the debugging option during development of your program and later turn it off. It could be inconvenient for you to have to set all the `LINK=` parameters in your source file and then later have to change them again. A pair of global equates are provided so you can simulate the `LINK=` parameter without explicitly specifying the parameter on each macro call.

The global `LinkAll` is assumed to be 0 (false). If you set it to 1 (true), then `LINK=Y` will be assumed on all the `PROCEDURE` and `FUNCTION` calls. The second global is named `Debug`. It too is initially 0. But by setting it to 1 you simulate `LINK=DEBUG`. Note that the `Debug` setting overrides the `LinkAll` setting. Both cause the `LINK/UNLK` pair to be generated, but the `Debug` global generates the module name after the exit code for `MacBug`. The setting of these globals should be done prior to using any of the program structure macros. You may set them explicitly in your source via `EQU` or `SET` statements or through the Assembler's `-d` command line option.

The general skeleton for the code generated by just the `PROCEDURE` or `FUNCTION` macros is as follows:

```

module-id          PROC | FUNC  ENTRY | EXPORT  ; Start a new code module
                    RECORD      {FramePtr},DECR ; and declare its scope
SF#xxxx            ; Origin-shifted
                    ; decremting stack frame
[resultName       DS.size      0 ]           ; Function result if this
                    ; is a function
formali           DS.size      amount       ; Declare all formals...if
                    ; any (all are word
                    ; aligned)
RetAddr            - - -                   ; i=1 to n or n to 1 for C
                    DS.L              1     ; Return address

```

The stack frame template has a unique name, `SF#xxxx` (the `x`'s are digits), for each procedure. It is a decremting, origin-shifted template with the origin defined by the field `FramePtr`. The field `FramePtr` is generated either by the first `VAR` declaration or `BEGIN` if there are no local variables declared (the descriptions of each of these macros will show the skeletons they follow in completing the stack frame). The template definition is left open by the `PROCEDURE` and `FUNCTION` macros because either a `VAR` or `BEGIN` macro must follow. `VAR` macros are used to define local variables that will be added to the stack frame. The `BEGIN` macro actually closes the template definition.

---

## Local variable declaration

`VAR local,...`

where

`local ::= formal` (see "Procedure or Function Header," given earlier in this appendix)

The `VAR` macro is used to declare local stack variables that are to be used by the procedure. Each `VAR` macro may declare one or more variables. `VAR` macro calls must be placed after the procedure or function header and before the `BEGIN` macro.

The syntax for defining local variables is exactly the same as that for declaring formal procedure parameters (see "Procedure or Function Header," given earlier in this appendix). Each variable is added to the stack frame template definition opened by the `PROCEDURE` or `FUNCTION` macro the same way formal parameters are. The only difference is that local variables are not automatically aligned to word boundaries and rounded up to an even size.

The general skeleton for the code generated by `VAR` macros continues the skeleton started in the procedure and function header description given earlier.

```
LinkA6    DS.L      1          ; A6 link to previous stack frame
FramePtr  EQU      *          ; Stack frame Origin
                          ; (proc's A6 points here)

locali    DS.size  amount    ; Declare a local.
                          ; (one for each VAR parameter)
- - -
```

On the first `VAR` call following the procedure or function header, the fields `LinkA6` and `FramePtr` are generated. They are followed by the local variable definitions. Each additional `VAR` call just adds its variable definitions to the stack frame.

When local variables are declared, it is assumed a `LINK` on `A6` will be generated (by the `Begin` macro). Thus the `LinkA6` field is generated to hold the previous stack frame link address. The frame pointer will be defined as `A6` and will point to `LinkA6` (see "Procedure or Function Start," given later in this appendix). This then becomes the origin for the template definition. `FramePtr` was defined as the origin field label when the template definition was opened (see procedure and function header skeleton), so it is now defined as the same offset as `LinkA6` (remember this is a decrementing template definition).

---

## Procedure or function start

```
BEGIN [non-blank] [,Save= reg-list] [ ,WITH = { template-id  
 ( template-id,... ) } ]
```

The BEGIN macro is placed after all the local variable declarations (if any). BEGIN is used to close (ENDR) the stack frame template definition begun by the PROCEDURE OR FUNCTION header macros and to generate the procedure's entry point code.

The general skeleton for the code generated by the BEGIN macro is shown here:

```
[ LinkA6    DS.L    1 ]      ; A6 link if required and allowed  
[ FramePtr  EQU    * ]      ; Stack frame Origin (only if no Var macros)  
  LocalSize EQU    *      ; Negative of the size of the stack frame  
      ENDR          ; Close the stack frame template definition  
      WITH    template-id,... ,SF#xxxx  
              ; Cover WITH= templates and the stack frame  
      [ LINK    A6,#LocalSize ]  
              ; Allocate local stack frame if required  
              ; and allowed  
      FP      SET    A6 or A7 ; A6 if LINK generated else A7  
      [ MOVE[M].L    reg-list,-(A7) ]  
              ; Save registers specified by Save=
```

If there are no local variables declared, the LinkA6 and FramePtr fields are generated (see "Local Variable Declaration," given earlier in this appendix, for the meanings of these fields). The BEGIN macro signals the end of the local variable declarations and defines procedure entry point, so that the stack frame definition is closed with LocalSize set to the current stack frame offset. Since the stack frame is defined as a decrementing template, LocalSize is a negative value whose absolute value is the size of the template. LocalSize is used for the LINK instruction.

The LINK instruction is generated under the following conditions:

- There are local variables declared with VAR macros.
- You specify register to be saved with the SAVE= parameter of BEGIN, and you have either a function with a return value (*result* specified on a PROCEDURE OR FUNCTION macro), or formal parameters declared on a PROCEDURE OR FUNCTION macro.
- LINK is forced by the LINK={Y | DEBUG} parameter on a PROCEDURE OR FUNCTION macro.
- LINK is forced by setting either of the globals, LinkAll or Debug, to 1 (true).

Even if you have any of these conditions you may still suppress LINK by specifying any nonblank value as the first parameter to BEGIN.

If none of the listed LINK conditions is true, or you suppress LINK with the nonblank first BEGIN parameter, then the stack frame pointer, FP, will be defined to be A7. If LINK is generated, FP is defined as A6. You use the frame pointer register to access the local variables and arguments on the stack. The variables may be directly referenced as frame pointer offsets, for example, *local(FP)*.

The stack frame field names may be directly referenced because the BEGIN macro always generates an Assembler WITH directive to cover the stack frame template. You may add your own template and data module names to this WITH for additional coverage by specifying the WITH= BEGIN parameter. If more than one name is specified, you must enclose the list in parentheses. Note in the skeleton that the stack frame name is always placed last in the WITH list so its definitions will override fields that belong to templates mentioned earlier in the list.

You may specify that registers are to be saved on the stack by using the SAVE= BEGIN parameter. This parameter takes either a single register or a list of registers using the same syntax defined for a MOVEM instruction. The registers that are saved will be restored for you when you exit from a procedure via a RETURN macro.

---

### Procedure or function secondary entry point

*entry-id* ENTER [*non-blank*] [ , WITH = { *template-id*  
(*template-id*,...) } ]

Sometimes it is convenient to have one procedure or function with multiple entry points; for example, a sine function with a cosine entry point. You can use the ENTER macro to define a secondary entry point into a procedure module.

ENTER is used within the body of a procedure (that is, somewhere beyond the BEGIN call). A label, *entry-id*, should be specified to define the secondary entry point. It is up to you to define its scope; that is, EXPORT or ENTRY (the default) prior to calling ENTER.

The parameters of ENTER are the same as those of BEGIN and perform the same functions. Thus you may define additional WITH coverage and suppress LINK. However, the one parameter missing in ENTER that is in BEGIN is the list of registers to be saved. The registers to be saved are assumed to be the same as those specified by BEGIN (there can only be one set of saved registers so that RETURN knows how to restore them).

The general skeleton of the code generated by ENTER is shown here:

```

entry-id    BRA.S %L%x%x%x    ; Branch around secondary entry point code
            ; Entry point label -- you define its scope
[           WITH            template-id,... ]
            ; Cover additional WITH= templates
[           LINK            A6,#LocalSize ]
            ; Allocate the local stack frame
[           MOVE[M].L      reg-list,-(A7) ]
            ; Save registers specified by
            ; Save= on Begin macro
%L%x%x%x    ; The branch-around label (the x's are digits)

```

A short branch is always generated around the entry point code. WITH, LINK, and MOVE[M] are generated just as they are in BEGIN (except that WITH does not respecify the stack frame).

---

## Procedure or function exit

```

RETURN [ret-result]
where
ret-result    ::=    ea[:sz]
sz              ::=    B|W|L|S|D|X|P

```

At the end of the procedure you must generate code to return to the caller. The RETURN macro is used to perform this function. It is responsible for

- restoring any saved registers
- unlinking (UNLK) the stack frame if LINK was generated on entry
- popping the parameters off the stack (if not C)
- leaving a function result on the top of the stack (if not C)
- returning to the caller
- defining the module name as a string for MacsBug

If you are exiting from a C procedure (as indicated by the C parameter on the PROCEDURE and FUNCTION macros), or if you are exiting from a procedure that has no arguments and is not a function with a result to return, then the following skeleton illustrates the code generated by RETURN:

```

[ MOVE[M].L  (A7)+, reg-list ]    ; Restore registers specified by
                                ; Save= on Begin macro
[ UNLK      A6 ]                  ; Set A6 back to the caller's stack frame
                                ; Exit
[ DC.B      'module-id' ]        ; Name for MacsBug as an as-is string
                                ; (8 chars)

```

This skeleton shows the basic exit code. Registers saved by the `BEGIN` macro at entry are restored, `UNLK` is generated if `LINK` was generated by `BEGIN`, and execution exits. If you specified the `LINK=DEBUG` parameter to the `PROCEDURE` or `FUNCTION` macros, or you defined the global `DEBUG` as 1 (`TRUE`), then the module name is generated for MacsBug following `RTS`. The name will be generated as an as-is string of eight characters, padded on the right with blanks if necessary.

If you are not exiting from a C procedure and the procedure has arguments or there is a result to return, as specified by the optional *ret-result* parameter of `RETURN`, then the following skeleton illustrates the code generated:

```
[ MOVE [M].L (A7)+,reg-list ]           ; Restore registers specified by
                                        ; Save= on Begin macro
[ UNLK A6 ]                             ; Set A6 back to the caller's stack
                                        ; frame
MOVEA.L (A7)+,A0                         ; Pop return address into A0
ADDQ.W #argsize,A7                       ; Pop arguments off the stack
                                        ; (argsize ≤ 8)

      or

LEA   argsize(A7),A7                      ; Pop arguments off the stack
                                        ; (argsize > 8)
[ [F]MOVE.sz ea,(A7) ]                  ; Set function value (MOVE if sz ::= B |
                                        ; W | L)
      JMP (A0)                            ; Exit
[ DC.B 'module-id' ]                    ; Name for MacsBug as an as-is string
                                        ; (8 chars)
```

As in the simple case, the registers are restored, `UNLK` is generated, and the name for MacsBug is generated. But if there are arguments, they must be popped off the stack. They are popped by generating `ADDQ` or `LEA` to pop the stack most efficiently. If a function result is specified with the *ret-result* parameter, then it is left on the top of the stack prior to returning to the caller.

Note that *ret-result* may only be specified if *result* was specified on the `PROCEDURE` or `FUNCTION` macro. The default size for the move is taken from the size specified for the `PROCEDURE` or `FUNCTION` *result*. If an identifier is specified for *result* instead of a size, then word is assumed. However, you do not have to use the size specified on the `PROCEDURE` and `FUNCTION` macros. Instead it may be overridden with an explicit size specification following the effective address portion of the *ret-result* (see syntax at the beginning of this section). No matter which way the size is specified, a standard `MOVE` is generated when the size is byte, word, or long (`B`, `W`, or `L`) and a `FMOVE` when the size is single, double, extended, or packed (`S`, `D`, `X`, or `P`).

There is no requirement that a function's result must be set on exit from the module with `RETURN`. It may be more convenient at times to set the return value at various places within the module and just use `RETURN` for a common exit point. When a function *result* is specified to the `PROCEDURE` and `FUNCTION` macros, a field is defined in the stack template that corresponds to where the function result is to be returned; that is, the first field of the stack frame template with no space reserved, which corresponds to the stack position after all the arguments and `RETURN` have been popped off the stack. It may be addressed through the field name. This name is the same as the *module-id* if you specified the *result* as a size, or if an identifier was specified instead of a size, that identifier is used as the field name. You may access it prior to exit just like any of the other stack frame fields, namely as an offset from the frame pointer register, `FP`; for example, `MOVE .w DO, name(FP)`.

---

### Procedure, function, or trap invocation

```
call[. ext] module-id[: result-sz] [([arg],...)][, disposition]
```

where

```
ext           ::= S | B | W | L | *
result-sz    ::= B | W | L
arg          ::= ea[: arg-sz] | NIL | TRUE | FALSE
arg-sz       ::= B | W | L | A | D-register
disposition ::= PASS | (ea | CC | POP)[: result-sz] | (PASS, (ea | CC)[: result-sz])
```

The `CALL` macro is used to invoke a procedure, function, trap, or another macro. Arguments may be pushed on the stack. If the invoked routine is a function, space on the stack may be reserved for the function result and the disposition of that result may be specified. The following general code skeleton shows how all these actions are performed:

```

[ SUBQ.W      #2 or #4,A7 ] ; Leave space if result-sz specified
CLR.L        -(A7)         ; If NIL
ST           -(A7)         ; If TRUE
CLR.B        -(A7)         ; If FALSE
MOVE.arg-sz  ea, -(A7)   ; If ea:arg-sz, where arg-sz ::= B | W | L
PEA         ea           ; If ea:A
MOVEQ       ea, D-register ; If ea:D-register...
MOVE.L      D-register, -(A7) ; ...ea:D-register generates a push of a long
- - - ; Additional arguments are pushed onto
; the stack
JSR         module-id    ; Call if external or ext ::= *
      or
BSR.ext    module-id    ; Call if ext ::= S | B | W | L
      or
module-id ; Call if module-id is opword, macro, or _id
TST.result-sz (A7)+      ; If result-action ::= CC:result-sz
      or
ADDQ.W      #2 or #4,A7   ; If result-action ::= POP:result-sz
      or
MOVE.result-sz (A7)+, ea ; If result-action ::= ea:result-sz
      or
TST.result-sz (A7)      ; If result-action ::= (PASS, CC:result-sz)
      or
MOVE.result-sz (A7), ea ; If result-action ::= (PASS, ea:result-sz)

```

Arguments are passed on the stack by enclosing the argument list in parentheses after *module-id*. An argument may take any of the following forms:

- The identifier `NIL`. `NIL` causes a push of a long word 0 by generating `CLR.L -(A7)`.
- The identifier `TRUE`. `TRUE` causes a word push of `$FFxx` (the `xx` is a garbage byte) by generating `ST -(A7)`.
- The identifier `FALSE`. `FALSE` causes a word push of `$00xx` (the `xx` is a garbage byte) by generating `CLR.B -(A7)`.
- An effective address optionally followed by a size (`B`, `w`, or `L`) specifier. The size specifier indicates the size of the argument that is to be pushed on the stack (byte, word, or long). `MOVE.sz ea, -(A7)` is generated. If no specifier is given, a word push is assumed. Note that `MOVE.sz #0, -(A7)` will be optimized by the Assembler to `CLR.sz -(A7)` unless `OPT NOCLR` or `OPT NONE` is in effect.
- An effective address followed by the address specifier `A`. Using `A` as a specifier indicates that the address of the argument is to be pushed (`PEA`).



- An effective address followed by a D-register. This is a special case that restricts the effective address to an immediate mode (that is, #ae) with a value -128..+127. It is used to generate a space-efficient long word push of small constants by generating MOVEQ to the specified D-register and then a move of the D-register onto the stack.
- An argument in the argument list may be omitted, indicated by a comma with nothing before it. Nothing is generated for such an argument. This is useful for commentary purposes when the first argument is already on the stack; for example, F(, Y). It could also be used for arguments beyond the first argument if, for example, you push a long argument that actually corresponds to two word arguments. Thus \_MOVETO QuickDraw call, which requires integer point coordinates as parameters, could be written as \_MOVETO(h,v). But if you happen to have a template defined that overlays the two integer coordinates with a long word, called, for example, thePoint, you could write \_MOVETO(thePoint:L,).

To call the specified *module-id* either JSR, BSR, or *module-id* itself is used to perform the call. The determination of which call form is used is based on the following criteria:

- JSR is generated whenever the routine you are calling is either undefined, a code module previously declared in the same file, or a code import; or an asterisk (\*) is specified for the CALL extension attribute, *ext*, to force JSR.
- BSR is generated if you specify an explicit extension, s or B, w, or L, indicating the size of BSR. If you don't specify the extension, BSR (default size) is still generated if *module-id* does not satisfy the JSR criteria or the following criteria.
- If *module-id* is an opword (defined by the Assembler's OPWORD directive), or a MACRO name, or an undefined type (as determined by the Assembler's &TYPE function) and begins with an underscore, then *module-id* itself is used as an opcode. This will cause generation of the opcode word (usually a trap) for opwords and a macro call for macros.

If a function is being called, then the amount of space to be reserved on the stack for the function result is specified with *result-sz* following *module-id*. The result size may B, w, or L, with both B and w reserving a word on the stack and L reserving a long word.

On return from the function it is assumed the result is left on the top of the stack in the space reserved prior to the call. The action to perform on the function result is specified by *disposition*. The value of *disposition* may be just the keyword PASS, an effective address, or the keywords CC or POP optionally followed by a size attribute.

Each of these forms causes a different action to be performed, as follows:

- `PASS` is the default disposition. This leaves the function result on the top of the stack.
- `CC` causes the function result to be tested by a `TST` instruction to set the condition codes.
- An effective address, *ea*, indicates that the function result is to be copied to the specified alterable address.
- `POP` indicates that the function result is to be popped off the stack.

Note that specifying an effective address, `CC`, or `POP` all cause the function result to be popped off the stack. The size attribute, *result-sz*, indicates how much to pop off the stack. The default is to use the size specified by *result-sz*, which is the normal case. The only time you might want to use a different pop size is if, for example, you want to pop less than what is actually the result size, leaving a portion of the result on the stack. For example, this could be used for the Menu Manager's `MenuSelect` call as follows:

```
CALL _MenuSelect:L(thePoint:L),D0:W
```

Since `MenuSelect` returns a long word with the high-order word containing the menu ID and the low-order word containing the menu item number, you could pop just the menu ID off the stack into `D0` while leaving the item number on the top of the stack for later use.

A situation exists where you may want to copy the function result or set the condition codes based on the result, but still want to leave the result on the top of the stack. By enclosing the keyword `PASS` followed by an effective address or `CC` in parentheses, you may perform this action. The value of *result-sz* for this case indicates the size of the `MOVE` or `TST` instruction.

---

## Considerations for use

This section describes the various considerations you need to take into account to use the structured assembly macros. The considerations are split into two groups. The first are the more esoteric considerations: why you should or should not use the macros. The second group of considerations are specific and cover the rules you must follow to use the macros, over and above the syntax and semantic considerations described in the previous sections.

---

## Why you should or should not use the structured assembly macros

This document provides a set of macros that essentially simulate almost all the functionality provided by a high-level language compiler. You have many of the advantages of a higher-level language but still retain the ability to “drop” into assembly code at any time. This is a powerful combination, and you can be reasonably sure that the code generated is as optimal as you can write.

However, you must remember that this “compiler” is implemented as a set of Assembler macros. Macros are more or less interpreted, and that means that this compiler is an interpretive compiler. In terms of raw speed it will never compile and generate code as fast as a native code compiler. The MPW Assembler is fast, but there is a limit. The macros presented here are extremely complex and push the Assembler to that limit. Average compilation speed is about 10,000 lines per minute on a 1-megabyte Macintosh Plus. That may appear faster than most compilers, but remember that most of those “lines” are macro definition lines, not source input lines. The actual net throughput is more on the order of 500–800 source lines per minute.

Error and consistency checking is not perfect, and is in fact almost nonexistent, because such checking would slow the macros down even more (*much* more). Except for a few tests for specific parameter syntactic forms and depending on the Assembler itself to catch errors, there are no additional error checks.

You have to weigh the advantages of using these macros against the assembly speed disadvantages. The reason the assembly speed aspect is noted as a consideration at all is that users of the Assembler not using these macros will see a large decrease in assembly speed when they do. This can be very disconcerting and discourage use of the macros, especially if you are already used to the speed of the Assembler and you know how long it takes to assemble a file. Those already coding in assembly language also have their own styles of coding and these macros will greatly affect those styles. It is conceivable that any advantages gained by using the macros cannot be justified when weighed against the loss of assembly speed or change of coding style.

Of the two groups of macros, program structure macros and flow-control macros, the flow-control macros are the more complex and require the most assembly time. You could decide to compromise and use only the simpler program structure macros. If you consider ease of coding, maintainability, and readability more important than assembly time, then you may want to use the full set of macros.

These arguments are presented for your careful consideration. Only you can decide whether these macros are worth using.

---

## Rules for using structured assembly macros

It is assumed in the following discussion that you already know how to use the Assembler and its directives and modes, and how to call macros.

The following rules must be observed when using the macros:

- Nesting of `IF`, `SWITCH`, `REPEAT`, `WHILE`, and `FOR` statements is limited to a maximum depth of 25.
- To use the flow-control macros you must have `BLANKS ON` (the Assembler default setting). The flow-control macros "read" like high-level language statements, with spaces or tabs between the keywords.
- The macros assume `OPT ALL` is in effect because they generate generic instructions and depend on the modes of their effective address parameters to cause the proper instruction generation. Indeed, code generated from macros is one of the prime reasons for having generic instructions in an Assembler.
- Don't generate assembly listings with `PRINTGEN` in effect. (This is a recommendation, not a must.)
- Macro parameters may be continued the usual way using the standard Assembler "`\`" continuation character. This could be particularly useful for the `VAR` program structure macro for declaring multiple local variables and commenting on them with one `VAR` declaration.
- Keyword macros may be in any order and there need not be a comma preceding the first keyword parameter if all the positional parameters are omitted. These are standard Assembler macro rules, but it is important to point out the fact that the syntax descriptions do not show all possible keyword permutations. For example, the `BEGIN` program structure macro may have its nonblank first (and in this case, only) positional parameter omitted. If it is omitted, the comma that follows it may be omitted and the two keyword parameters could be reversed.
- Long branches (produced from an `L ext` extension specification) may only be used if the `MC68020` or `MC68030` is specified as the target microprocessor.
- Branch warnings are not suppressed. This is done so you can get the appropriate extension specifications on the macros that require them, and normally default to word size branches.
- Due to the size of the structured assembly macros, you cannot use them on a 512K machine.

The macros are provided as two source files ready for assembly in the folder AStructMacs. File ProgStrucMacs.a contains the program structure macros and file FlowCtlMacs.a contains the flow-control macros. The folder also contains Sample.a, which is a rewrite of Sample.a in the AExamples folder, except that it illustrates the use of the structured macros. Both macro files are set up to generate DUMP files (ProgStrucMacs.d and FlowCtlMacs.d). You have the choice of editing the source files to remove the DUMP directives or just assembling them to generate the DUMP files. It is recommended that you generate DUMP files because to use the sources you have to INCLUDE them. Considering the size of these source files it is not a good idea to INCLUDE them. LOAD statements are very much faster.

No matter which technique you decide to use, it is suggested that you put the files in the directory specified by the {AIncludes} MPW Shell variable. Since the {AIncludes} Shell variable is known to the Assembler as part of its standard search rules for input file pathnames, you only have to LOAD or INCLUDE the files with no additional qualification and no -I Assembler option to access them.

---

## Syntax summary

---

### Expressions

---

```
expr ::= s-expr | s-expr op s-expr
s-expr ::= cc | ea cc [.sz] ea
op ::= AND | OR
cc ::= EQ | NE | LE | LT | GE | GT | MI | PL | HI |
      LS | LO | CC | CS | NZ | HS | VC | VS
sz ::= B | W | L
```

---

## Flow-control macros

```
IF# expr1 THEN[. ext]
    statements1
    ELSEIF#[. ext] expr2 THEN[. ext2]
        statements2
    ELSE#[. ext]
        statements3
ENDIF#

SWITCH# [. sz] selector [,Dreg=Dn] [,JumpTbl={N|Y|ext}] [,ChkRng={N|Y}]

CASE#[. ext] ae1 [...ae2] ,...
    statements1
    DEFAULT#
        statements2
ENDS#

REPEAT#
    statements

UNTIL#[. ext] {expr | FALSE}

WHILE# {expr | TRUE} DO[. ext]
    statements

ENDW#

FOR# cl-var [= [. sz] initial] [DOWN]TO final [BY increment] [UNTIL expr] DO[. ext] \
    [,DREG=Dn] [,Opt={Y|N}] [,Clr={X|N}]
    statements

ENDF#

LEAVE#[. ext] [label] [IF[#]expr]

CYCLE#[. ext] [label] [IF[#]expr]

GOTO#[. ext] [IF[#]expr THEN[. ext]]label
```

---

## Program structure macros

```
ENTRY
EXPORT PROCEDURE [module-id][([formal,...]):result][,C] [,Link={Y|DEBUG}]\
LOCAL FUNCTION [,Main={N|Y}]
```

```
    formal ::= id:formal-sz[count]
    formal-sz ::= B|W|L|S|D|X|P|ae|template-id
    count ::= [ae]
    result ::= B|W|L|S|D|X|P|id
```

```
Var local,...
```

```
    local ::= formal
```

```
BEGIN [ non-blank ] [,Save = reg-list] [ ,WITH = { template-id
    ( template-id,... ) } ]
```

```
entry-id ENTER [ non-blank ] [ ,WITH = { template-id
    ( template-id,... ) } ]
```

```
Return [ret-result]
```

```
    ret-result ::= ea:sz
    sz ::= B|W|L|S|D|X|P
```

```
Call [,ext] module-id:result-sz [(arg,...)] [,disposition]
```

```
    ext ::= S|B|W|L|*
    result-sz ::= B|W|L
    arg ::= ea:arg-sz | NIL | TRUE | FALSE
    arg-sz ::= B|W|L|A|D-register
    disposition ::= PASS | ea | CC | POP[:result-sz] | (PASS,(ea | CC)[:result-sz])
```





## Glossary

**@-label:** A label of very limited scope, written with @ as its first character.

**absolute:** Of an expression, having a value that can be determined during assembly.

**actual parameter:** A parameter in a macro call.

**addressing mode:** see **effective addressing mode**.

**anonymous:** Of a file or variable, not having an identifier. Anonymous objects are accessed by pointers.

**application globals area:** An area in RAM in which application programs store data.

**application parameter area:** An area in RAM pointed to by register A5.

**array:** A data structure containing an ordered set of elements.

**ASCII:** Acronym for American Standard Code for Information Interchange; a system of assigning code numbers to letters, numerals, punctuation marks, and control codes.

**ASCII character:** A character whose ASCII code number lies in the range 0..127.

**as-is string:** A string that the Assembler stores without length information.

**assembler:** A program that translates source text into object code.

**Assembler option:** An instruction passed to the Assembler at the time it is invoked.

**Assembler system variable:** A variable whose value is determined by the Assembler.

**assembly:** The process of translating source text into object code. Also, the set of modules being assembled.

**assembly-control directive:** A directive that controls whether or not some portion of source text is assembled.

**backquote:** The ASCII \$60 character, written, which tells the Macro Processor that the next character is to be processed literally during macro expansion.

**binary:** Base-2 number representation, using the numerals 0 and 1.

**blank:** A tab or space character.

**body:** In a macro definition, the machine instruction and directive statements that comprise the macro, other than the MACRO directive, prototype line, and ENDM directive.

**Boolean expression:** An expression whose value is either true or false.

**built-in function:** A function that is part of the macro language.

**code:** Executable computer instructions.

**command line:** The text you enter to execute a command in the MPW Shell.

**comment:** Source text intended for a human reader, ignored by the Assembler.

**comment field:** The area of a source text line reserved for comments.

**comment line:** A source text line containing only a comment.

**comparison operator:** An operator that compares two values; they are listed in Table 2-3.

**conditional assembly:** The process of controlling what source text is assembled.

**continuation character:** A character at the end of a source text line that lets you continue the text onto a second line. Backslash (\) is the Assembler's continuation character.

**C string:** A string in a format compatible with the C programming language; it is terminated with a 0 byte.

**data:** Information that a computer processes.

**decimal:** Base-10 number representation, using the numerals 0 through 9.

**destination:** An address into which an instruction places data.

**diagnostic output file:** A file to which MPW tools, including the Assembler, write error messages and progress information.

**dimension:** An ordering relation among elements of an array.

**directive statement:** A source text instruction to the Assembler, which generates no direct code.

**dynamic nesting level:** The ordinal position of a macro call in a macro call chain.

**effective addressing mode:** Any of several ways of writing an address so it can be assembled.

**element:** One item in a sublist.

**equal:** (Of strings) indistinguishable.

**equate:** An EQU or SET directive.

**error code:** A code (usually a number) returned by the Assembler to indicate that it has encountered an error in the source text or assembly process.

**expand:** To replace a macro call with the appropriate macro body, substituting actual values for variables and parameters where needed.

**export:** To make a module or entry point defined in the current assembly linkable to other assemblies.

**field:** A data structure within a template or statement.

**first-level call:** The outer macro call of a macro call chain.

**floating-point:** A way of representing decimal numbers.

**formal parameter:** A parameter in a macro definition.

**generic form:** A way of writing a statement so that the Assembler will convert it into a different form.

**global scope:** The scope of code or data that is accessible in more than one module.

**global symbol table:** A symbol table that contains symbols with global scope.

**header:** In a macro definition, the MACRO directive itself.

**hexadecimal:** Base-16 number representation, using the numerals 0 through 9 and the letters A through F.

**identifier:** A name in source text.

**IF...ENDIF construct:** Source text enclosed by an IF directive followed by an ENDF directive.

**import:** To make a module or entry point defined in another assembly linkable to the current assembly.

**include:** To insert the contents of a source text file into an assembly.

**index:** A numeric value that indicates the position of an element in a sublist or array, expressed by a subscript.

**initial value:** The value the Macro Processor assigns to a SET variable when it is created.

**inner macro call:** A macro call inside a macro.

**instruction:** A program element representing a single computer operation.

**jump table:** In the Macintosh, a table of references in RAM, used for communication between segments.

**keyword:** An identifier for a macro parameter.

**keyword macro:** A macro whose parameters are identified by keywords.

**keyword macro call:** A call to expand a keyword macro.

**label:** A name that identifies a location in assembled code or data.

**label field:** The leftmost field of a statement.

**lifetime:** Of a variable, the duration of program execution during which it is accessible.

**literal:** Immediate data given to an instruction.

**literalize:** To modify a symbol in source text so that the Assembler does not interpret it.

**literal pool:** A table of all literals in an assembly, without duplications, maintained by the Assembler.

**local scope:** The scope of code or data that is accessible in only one module.

**local symbol table:** A symbol table that contains symbols with local scope.

**location counter:** A counter maintained by the Assembler that points to the current byte of code or data being assembled.

**logical operator:** One of the operators OR, XOR, AND, and NOT.

**machine instruction statement:** A statement that generates executable code.

**Macintosh library routine:** Any of the standard Macintosh routines described in *Inside Macintosh*.

**macro:** Defined source text that the Macro Processor expands into other source text on command.

**macro call:** A command to the Macro Processor to insert a macro at a specific point in a source text.

**macro call chain:** A sequence of one outer macro call and any number of inner macro calls.

**macro definition:** The source text that constitutes a macro.

**macro directive:** A directive that controls macro expansion but does not generate any source text directly.

**macro label:** A label indicating a location in source text, used only with GOTO directives.

**macro language:** The directives and functions that create and manipulate macros.

**Macro Processor:** The part of the Assembler that interprets the macro language.

**macro symbol table:** A symbol table that contains macro symbols and definitions.

**macro variable:** A variable whose value is controlled by macro directives.

**main code module:** The code module that will be executed first when a program runs.

**main data module:** A specific data module that the Linker places at the base of the global data area, where its offset is known to the Assembler.

**mixed-mode macro:** A macro containing both positional and keyword parameters.

**mnemonic:** A source text symbol that expresses an instruction or directive.

**model statement:** In a macro body, a statement that the Macro Processor uses as a model to generate actual machine instruction or directive statements.

**module:** A contiguous collection of code or data.

**module directive:** A directive that defines a code or data module.

**nonterminal symbol:** Part of a syntax diagram that stands for something to be written in the source text, such as *expr* for an expression.

**numeric constant:** A number expressed directly in a source text.

**object code:** Machine-language code generated by the Assembler.

**object file:** A file containing specifications for data and code module contents, references to other code and data modules, and segmentation information.

**operand field:** The source text column that contains instruction operands and directive parameters.

**Operating System routine:** A Macintosh library routine that performs a task such as accessing a disk file or handling an error.

**operation:** A directive or instruction in a source text.

**outer macro call:** A macro call that is not inside a macro.

**paired brackets:** A left bracket and a right bracket that the Assembler treats as enclosing an expression.

**paired parentheses:** A left parenthesis and a right parenthesis that the Assembler treats as enclosing an expression.

**paired single quotation marks:** The first and last single quotation marks in a quoted string.

**parameter:** A piece of data in the operand field of a directive, macro prototype, or macro call statement.

**parameter list:** A sequence of parameters, separated by commas, in a macro prototype or macro call.

**Pascal string:** A string assembled into a form compatible with the Pascal programming language; it begins with a length byte and has a maximum length of 255 characters.

**pass:** One processing cycle of the Assembler.

**PC-relative:** Located relative to the current value of the program counter.

**positional macro:** A macro whose parameters are identified by their position in its parameter list.

**positional parameter:** A macro parameter identified by its position in the parameter list.

**postfix notation:** A way of representing object code in an incomplete form, used by the Assembler on its first pass.

**predefined SET variable:** A SET variable whose value is set by the Macro Processor.

**prototype:** In a macro definition, the statement that establishes the name and parameter format of the macro.

**qualified:** Said of an identifier that is written with a qualifier.

**qualifier:** An identifier appended to another identifier (usually with a period between), which modifies its meaning.

**quoted string:** A string enclosed in single quotation marks ( ' ).

**register list:** A source text list of microprocessor registers, used as an operand of move-multiple instructions.

**relative ASCII ordering:** The algorithm by which the Macro Processor compares strings.

**relocatable:** Of an expression, having a value that cannot be determined during assembly.

**scope:** The area of source text in which a piece of code or data can be referenced.

**scratch file:** An area of RAM or disk memory used for temporary storage.

**second-level call:** The first inner macro call in a **macro call chain**.

**segment:** A collection of modules that is loaded together from disk into RAM during program execution.

**SET variable:** A macro variable whose value is assigned by a SETA or SETC directive.

**source:** In instruction syntax, an address from which the instruction takes data.

**source text:** Program text written by a programmer.

**statement:** A line of source text, including machine instruction statements and directive statements.

**string:** A sequence of one or more characters.

**string constant:** A string written explicitly in source text, enclosed in single quotation marks.

**sublist:** A collection of macro call parameters that the Macro Processor treats as a unit.

**subscript:** A numeric expression whose value is the index of an element in a sublist or array.

**symbol:** A lexical component of source text processed by the Assembler.

**symbolic parameter:** A variable that acquires a value during macro expansion.

**symbol table:** A list of source text symbols and their values maintained by the Assembler.

**table:** An ordered list of code or data objects that the Assembler creates in memory during assembly.

**template:** A source text structure that describes a collection of data without allocating memory for it.

**terminal symbol:** Part of a syntax diagram that must be written in the source text exactly as it appears in the diagram.

**token:** A character or group of characters that the Assembler interprets as a single syntactic entity.

**tool:** A program that runs in the MPW Shell environment.

**Toolbox routine:** A Macintosh library routine that performs a task such as creating a menu or manipulating a window.

**trailer:** In a macro definition, the ENDM or MEND directive.

**trap dispatcher:** A routine in RAM that handles unimplemented instructions, among other tasks.

**unequal:** Of strings, able to be distinguished.

**unimplemented:** Of an instruction, one that is not part of the standard MC68xxx instruction set.

**variable definition directive:** A directive that creates a macro variable.

**WHILE...ENDWHILE construct:** Source text enclosed by a WHILE directive followed by an ENDWHILE directive.

# Index

## Cast of Characters

& (ampersand) 124, 129, 144  
\* (asterisk) 24, 26, 121, 166  
@ (at sign) 17, 130, 213  
` (backquote) 130  
\  
(backslash) 23, 24, 149  
{ } (braces) xxi, 149  
[ ] (brackets) xxi, 128, 149  
: (colon) 107, 149  
, (comma) 149  
- (compliment) 29, 149  
/ (division symbol) 29, 148  
\$ (dollar sign) 26, 213  
... (ellipses) xxii  
= (equal sign symbol) 148, 175  
> (greater than symbol) 149, 175, 176  
≥ (greater than or equal to symbol) 29, 149, 175, 176  
>= (greater than or equal to symbol) 29, 149, 175, 176  
[ (left bracket) 149  
< (less than symbol) 175, 176  
≤ (less than or equal to symbol) 176  
<= (less than or equal to symbol) 29, 149, 175, 176  
\*\* (logical and) 29  
- (minus sign) 29, 148  
// (modulus division) 29, 148  
\* (multiplication) 29  
~ (NOT) 29, 149  
<> (not equal symbol) 29, 149, 175  
≠ (not equal symbol) 29, 149, 175  
# (number sign) 231  
( ) (parentheses) 29, 128  
% (percent sign) 26, 263  
.  
(period) 149, 213  
.\* (period-asterisk) 121  
+ (plus sign) 148  
" (quotation mark) 26  
} (right brace) 149  
] (right bracket) 149  
; (semicolon) 121  
<< (shift left symbol) 149  
>> (shift right symbol) 149  
' (single quotation mark) 128  
/ (slash) 29, 148  
\_ (underscore) 25, 213

## A

&ABS: return absolute value 146  
absolute expressions 31  
accessing variable substrings 158  
address  
  formats 192  
  optimizations 192  
  registers 44  
address syntax 35-56  
  ambiguities 41-43  
  forward-reference 43  
  MC68030 instructions 49  
  MC68851 instructions 53  
  MC68881 instructions 52-53  
  MC68882 instructions 52-53  
  modes 37-41  
  optimizations 41  
  registers 44-45  
  special address formats 46-48  
    literals 55-56  
    MC68xxx instructions 46  
    MC68020 instructions 47-48  
    MC68030 processor 49-51  
    MC68851 53-54  
    MC68881 and MC68882 instructions 52-53  
-addressize option 255  
AERROR directive 182  
anonymous module 65  
ANOP directive 182  
application global area 13  
application parameter area 13  
As-is string 28  
Assembler command syntax 238-259  
assemblers, comparison of 211-218  
  addressing 217  
  communicating between modules 215  
  defining modules 215  
  expressions, writing of 215  
  identifiers, writing of 213  
  location-counter reference 216  
  macros, writing of 217  
  module definition 215  
  number, writing of 214  
  strings, writing of 214

assembly files 7  
 assembly listing format 207-209  
 assembly options 93-100  
   BLANKS: control blanks in operand field 99-100  
   BRANCH and FORWARD: resolve forward branches 96-97  
   CASE: treatment of lowercase letters 98-99  
   MACHINE: specify target machine 93  
   MC68851: coprocessor instructions 95  
   MC68881 and MC68882: coprocessor instructions 94-95  
   OPT: specify level of optimization 97-98  
   STRING: specify format 95-96  
 @ labels 17, 130, 213  
 aware and nonaware tests 231-232

## B

backquote character ( ` ) 117, 130  
 backslash character ( \ ) 149  
 binary numbers 26  
 BLANKS directive 99  
 -blksize option 255  
 Boolean control expressions 175-176  
   comparing two integers 175  
   comparing two strings 175  
   comparing integers and strings 176  
 braces ( {} ) xxi, 149  
 brackets ( [] ) xxi, 128, 149  
 BRANCH directive 96  
 built-in functions 141-142

## C

CASE directive 98  
 C calling conventions 277-278  
   function results 278  
   parameters 278  
   register conventions 278  
 -c[heck] option 256  
 &CHR: convert integer to character 159  
 code and data module definitions, 62-67  
   CODE and DATA, switch between 67  
   END: end the assembly 67  
   FUNC and ENDFUNC, define function code module 63  
   MAIN and ENDMAN: define main program code module 63  
   PROC and ENDPROC: define procedure code module 62-63  
   RECORD and ENDR: define a data module 64-65, 76-80  
 CODE directive 67

CODEREFS directive 88  
 coding conventions 11-34  
   definitions, scope of 15-18  
   expressions 28  
     absolute 31  
     evaluation of 30  
     relocatable 30  
   imported and exported objects 17  
   machine instruction syntax 19-24  
     label field 20  
     operand field 23  
     operation field 21  
   segmentation 18-19  
   source text structure 13  
   symbols 25  
     identifiers 25  
     numeric constants 26  
     strings 27  
 command syntax 253-259  
 COMMENT directive 93  
 comments 24, 93  
 companion operators 29, 175  
 &CONCAT: concatenate strings 160  
 concatenating symbolic parameters 124  
 conditional-assembly directives  
   ACTR: limit looping 180-181  
   AERROR: error generation 182  
   ANOP: Assembler NOP 182  
   Boolean control expressions 175-176  
   CYCLE and LEAVE directives 180  
   EXITM and MEXIT: exit macro 181  
   GOTO and IF...: branching 176-177  
   IF, ELSEIF, ELSE, and ENDIF: conditional assembly 178-179  
   WHILE and ENDWHILE: looping 179  
   WRITE and WRITELN: write to output 181-182  
 condition codes 229-233  
 coprocessor instructions 94-95  
 C string 28  
 CYCLE directive 180

## D

data definition directives 59, 72-75  
   DC and DCB: place contents in code or data 73  
   DS: define storage area 75  
   data module definitions 64-67  
 DATA directive 67  
 DATAREFS directive 88  
 DC directive 73  
 DCB directive 73  
 DECREMENT parameter 65



&DEFAULT: return string value or default 160  
-d[efine] option 256  
definitions of code and data modules 62-68  
scope of 15-16

&DELSYMTBL: delete symbol table 157  
directives 57-112

ALIGN 100  
BLANKS 99  
BRANCH 96  
CASE 98  
CODE 67  
CODEREFS 88  
COMMENT 93  
DATA 67  
DATAREFS 88  
DC 73  
DCB 73  
DS 75  
DUMP 105  
EJECT 111  
END 67  
ENDFUNC 63  
ENDMAIN 63  
ENDPROC 62  
ENDR 64  
ENDWITH 82  
ENTRY 85  
EQU 68  
ERRLOG 106  
EXPORT 85  
FORWARD 96  
FREG 70  
FUNC 63  
IMPORT 87  
INCLUDE 104  
LOAD 105  
MACHINE 93  
MAIN 63  
MC68851 95  
MC68881 94-95  
OPT 97  
OPWORD 71  
ORG 102  
PAGESIZE 107  
PRINT 108  
PROC 62  
RECORD 64  
REG 70  
SET 68  
SPACE 111

STRING 95  
TITLE 108  
WITH 82

## E

ellipses xxii  
END directive 67  
ENDFUNC directive 63  
ENDMAIN directive 73  
ENDPROC directive 62  
ENDR directive 64  
ENTRY directive 85  
ENDWITH directive 82  
EQU directive 68  
equates 40, 68, 105  
-e[rrlog] option 257  
&EVAL: evaluate contents of string 147  
exclusive OK 29  
EXPORT directive 85  
expressions 28-33  
absolute 32  
evaluation of 30-31  
relocatable 33

## F

fields xx, 19-24, 76  
files, assembly-language 7  
search rules 104  
file control directives 103-107  
DUMP and LOAD: write and read symbol table 105  
ERRLOG: specify error log 106-107  
INCLUDE: take source text from another 104  
search rules 104  
&FINDSYM: find symbol in table 156  
flow-control macros 310  
-font option 110, 257  
formats 61-72  
FORWARD directive 96  
FREG directive 70  
FUNC directive 63  
FUNCTION macro 293, 311  
functions  
&ABS 146  
&CHR 159  
&CONCAT 160  
&DEFAULT 160  
&DELSYMTBL 157  
&ENTERSYM 155  
&EVAL 147  
&FINDSYM 156

&GETENV 160  
 &INTTOSTR 160  
 &ISINT 147  
 &I2S 160  
 &LC 161  
 &LEN 147  
 &LEX 148  
 &LIST 150  
 &LOWCASE 161  
 &MAX 151  
 &MIN 151  
 &NBR 151  
 &NEWSYMTBL 154  
 &ORD 152  
 &POS 152  
 &SCANEQ 153  
 &SCANNE 153  
 &SETTING 161  
 &STRTOINT 154  
 &S2I 154  
 &SUBSTR 162  
 &TRIM 163  
 &UC 164  
 &UPCASE 164  
 symbol table functions 154-157

## G

GBLA directive 143-145  
 GBLC directive 143-145  
 general assembly 196-200  
 generic instruction 185-188  
 &GETENV: return MPW Shell variable value 160  
 global symbol table 6  
 GOTO directive 178, 183  
 GOTO statement 176

## H

-h option 257

## I

identifiers 25  
 imported, exported objects 17  
 INCREMENT parameter 65  
 instruction sets 233-251  
   condition codes 229  
   instruction evaluation 225  
   instruction operands 214  
   instruction set listings 233  
   listing conventions 225-229

Cp type 228  
 flags 228  
 equivalent 229  
 group 228  
 operands 226-227  
 opcode 226-227  
 range 229

&INTTOSTR: convert integer to string 160  
 -i option 257  
 &ISINT: test string for integer content 147  
 &I2S: convert integer to string 160

## J

jump table 13

## K

keyword macros 135-137

## L

label field 20  
 &LC: convert string to lowercase 161  
 LCLA directive 143-145  
 LCLC directive 143-145  
 LEAVE directive 180  
 &LEN: measure string length 147  
 &LEX: parse string lexically 148  
 libraries, Macintosh 9  
 linker and scope controls 60, 84-93  
   CODEREFS and DATAREFS: control name linking 88  
   EXPORT and ENTRY: expand scope of entry points 85-87  
   SEG: specify current code segment 92  
 Lisa Workshop 211  
 &LIST: divide string into list 150  
 listing controls 107-111  
   EJECT: start new page listing 111  
   PAGESIZE: specify listing page size 107-108  
   PRINT: control listing information 108-111  
   SPACE: insert blank line 111  
   TITLE: specify title line 108  
 literals 55, 113, 196  
 local symbol table 6  
 location-counter controls 100-102  
   ALIGN: align location counter 100  
   ORG: set location counter 102-103  
 logical operators 29  
 -l option 255, 257  
 -lo option 258  
 &LOWCASE: convert string to lowercase 161

## M

**MACHINE** directive 93  
**machine instruction syntax** 19  
    comments 24  
    label field 20  
    operand field 23–24  
    operation field 21  
**Macintosh character set** 219  
**macros** 115–182  
    body 120  
    calling 125–134  
    call labels 127  
    comments 121–122  
    conditional-assembly directives 173–182  
        Boolean control expressions 175–176  
        **MACRO**, **ENDEM**, and **MEND** delimit 119  
        prototype statement 119  
        symbolic parameters 123  
    controls, scope of 118  
    defining 118–125  
        **MACRO** and **ENDM** delimit 119  
        prototype statement 119  
    expansion 117  
    keyword  
        calling 136–137  
        defining 135  
    mixed-mode 138  
    nesting macros 133–135  
    object assembler 261–269  
    operand sublists 131–133  
    operand syntax 128–131  
    symbol table 6  
        *See also* structured assembly  
main code module 15  
main data module 66  
**MAIN** parameter 66  
**&MAX**: find maximum integer in list 151  
**MC68000** 5  
**MC68010** 5  
**MC68020** 5, 47–49  
**MC68030** 5, 49  
**MC68851** 5, 53  
**MC68881** 5, 52–53  
**MC68881** directive 94–95  
**MC68882** 5, 52–53  
**MC68851** directive 95  
**&MIN**: find minimum integer in list 151  
mnemonics 21, 223  
model statement 120  
**MULS** 47, 192  
**MULU** 192

## N

**&NBR**: count sublist elements 151  
nesting macros 133–135  
notation conventions xviii  
    braces and brackets xxi  
    Courier typeface xviii  
    delimiter symbols xx  
    ellipses xxii  
    fields xx  
    italic xix  
    underlining xxii  
numeric constants 26–27

## O

object files 13  
object-oriented programming  
    **EndObjectWith** 266  
    **EndMethod** 267  
    **FuncMethOf** macro 267  
    **IMPL** keyword 265  
    **IMPL** macro 265  
    **Inherited** macro 268  
    **InitObjects** macro 263  
    **MethCall** macro 268  
    **MoveSelf** macro 269  
    **NewObject** macro 269  
    **ObjectDef** macro 263  
    **ObjectInf** macro 265  
    **ObjectWith** macro 266  
    **ProcMethOf** macro 267  
-o option 258  
opcode 226  
operand field 23  
operation field 21–23  
**OPT** directive 97  
optimization of instructions 22–23  
options 255–259  
    **-addrsize** option 208–209, 255  
    **-blksize** option 255  
    **-c[heck]** option 256  
    **-d[e]fine** option 256  
    **-e[rrlog]** option 257  
    **-font** option 110, 209, 257  
    **-h** option 257  
    **-i** option 257  
    **-l** option 255, 257  
    **-lo** option 258  
    **-o** option 7, 258  
    **-p** option 7, 258  
    **-pagesize** option 258

**-print** 114, 258  
**-s option** 259  
**-sym off** 259  
**-sym [on][full]** 259  
**-t option** 259  
**-w option** 259  
**-wb option** 259

option directives

**BLANKS:** control blanks in operand field 99-100  
**BRANCH, FORWARD:** resolve forward branches 96-97  
**CASE:** treatment of lowercase letters 98-99  
**MACHINE:** specify target machine 93  
**MC68851:** coprocessor instructions 95-96  
**MC68881, MC68882:** coprocessor instructions 94-95

**ORG directive** 102

**&ORD:** return integer value 152

**P**

**PAGESIZE directive** 107

**-pagesize option** 258

**Pascal calling conventions** 273-276

function results 275-276

parameters 273-274

real-type 274

structure-type 275

register conventions 277

**PASCAL string** 28

**&POS:** find substring 152

**-p option** 255

**print directive parameters** 109

**-print option** 258

**PROC directive** 62-63

**PROCEDURE macro** 293, 311

**programming for Macintosh** 8-9

**program structure macros** 281, 292-294, 311

**R**

**RECORD directive** 76

**REG directive** 70

**relocatable expressions** 31-33

**S**

**scope of definitions** 15-18

**SEG directive** 92

**segmentation of code** 18

**SET directive** 68

**SET variables** 141-143

**&SETTING:** return directive setting 161

**&SCANEQ:** scan string 153

**&SCANNE:** scan string 153

**s option** 259

**source text structure** 14

**SPACE directive** 111

**special address formats** 192-195

bit field instructions 193

**CAS, CAS2:** comparing and swapping 193

**DIVs, DIVU:** signed, unsigned division 193

**FMOVE with packed BCD data** 194

**FMOVEM with explicit register lists** 194

**FSINCOS:** simultaneous sine and cosine 194

**FTcc, FTPcc:** floating-point trap on condition 194

literals 195

for MC6800 192

for MC8020 192-193

for MC68881 and MC68882 194-195

for MC68851 195

**PACK, UNPK:** pack and unpack 193

**Tcc, TPcc:** trap on condition 193

**TDIVS, TDIVU:** truncated signed, unsigned division 193

**status codes** 7

**STRING directive** 95

**strings** 27-28, 152-155, 157-159

**&STROINT:** convert string to integer 154

**structured assembly macros** 279-311

expressions 281-282

flow-control macros 283-292

Cycle statement 291

For statement 288

GoTo statement 292

If statement 283

Leave statement 290

Repeat statement 287

Switch statement 285

While statement 287

**program structure macros** 292-305

code generation 294

local variable declaration 298

procedures and functions 295, 298-306

syntax 309-312

expressions 309

flow-control macros 310

program structure macros 311

usage, considerations for 306-308

**&SUBSTR:** return substring 162

**symbol definitions**

**EQU and SET:** name constants and registers 68

**OPWORD:** name machine instruction 71-72

**REG and FREG:** name register list 70

**symbols** 25

**-sym off** option 259  
**-sym [on] [off]** option 259  
syntax diagrams 189-204  
    assembly-language addresses 191  
        addressing modes 191  
        literals 195  
        macro 200  
        SET variable 202  
syntax rules: see coding conventions  
**&SYSDATE**: current date 170  
**&SYSFLAGS**: values set by **&FINDSYM** 171  
**&SYSGLOBAL**: symbol table IDs 171  
**&SYSINDEX**: macro call index 168  
**&SYSLIST**: macro operand list 169  
**&SYSLOCAL**: symbol table IDs 171  
**&SYSLSLST**: macro call index 168  
**&SYSMOD**: current module identifier 170  
**&SYSNDX**: macro call index 168  
**&SYSSEG**: current segment identifier 170  
**&SYSTIME**: current time 170  
**&SYSTOKEN**: values set by **&LEX** 171  
**&SYSTOKSTR**: values set by **&LEX** 171  
**&SYSVALUE**: values set by **&FINDSYM** 171

## T

template definitions 76-92  
    linker, scope controls 60  
    **RECORD** and **ENDR**: define a template 76-80  
    **WITH** and **ENDWITH**: supply record name qualification 82-84  
**TITLE** directive 108  
**-t** option 259

## U

underlining xxii

## V

variables 139-172  
    Assembler system variables 168-172  
    SET variables 141-144  
    **&SYSDATE**: current date 170  
    **&SYSFLAGS**: value set by **&FINDSYM** 171  
    **&SYSGLOBAL**: symbol table IDs 171  
    **&SYSINDEX**: macro call index 168  
    **&SYSLIST**: macro operand list 169  
    **&SYSLOCAL**: symbol table IDs 171  
    **&SYSLSLST**: macro operand list 169  
    **&SYSMOD**: current module identifier 164  
    **&SYSNDX**: macro call index 168  
    **&SYSSEG**: current segment identifier 170

**&SYSTIME**: current time 170  
**&SYSTOKEN**: value set by **&LEX** 171  
**&SYSTOKSTR**: value set by **&LEX** 171  
**&SYSVALUE**: value set by **&FINDSYM** 171

## W

**-wb** option 259  
**WITH** directive 82  
While statement 287  
**-w** option 259

## X

XOR operator 29

## THE APPLE PUBLISHING SYSTEM

This Apple® manual was written, edited, and composed on a desktop publishing system using Apple® Macintosh® computers and Microsoft® Word software. Proof and final pages were created on the Apple LaserWriter® II NTX printer. POSTSCRIPT®, the LaserWriter® page-description language was developed by Adobe Systems Incorporated. The illustrations were created using Adobe Illustrator. Some syntax diagrams were prepared using MathType.

The illustration on the cover was generated using Adobe Illustrator 88 on a Macintosh® II computer. Some of the images were scanned using an Apple® Scanner and then manipulated in Image Studio. Initial proofing was done using a QMS color printer. Color separations were done using Adobe separator and output to a Linotronic 300 at standard resolution.

Text type is Apple's corporate font, a condensed version of Garamond. Bullets are ITC Zapf Dingbats. Some elements, such as programs listings, are set in Apple Courier, a fixed-width font.

EQ

NE

OG

UL

OG

UL

C

U

OL