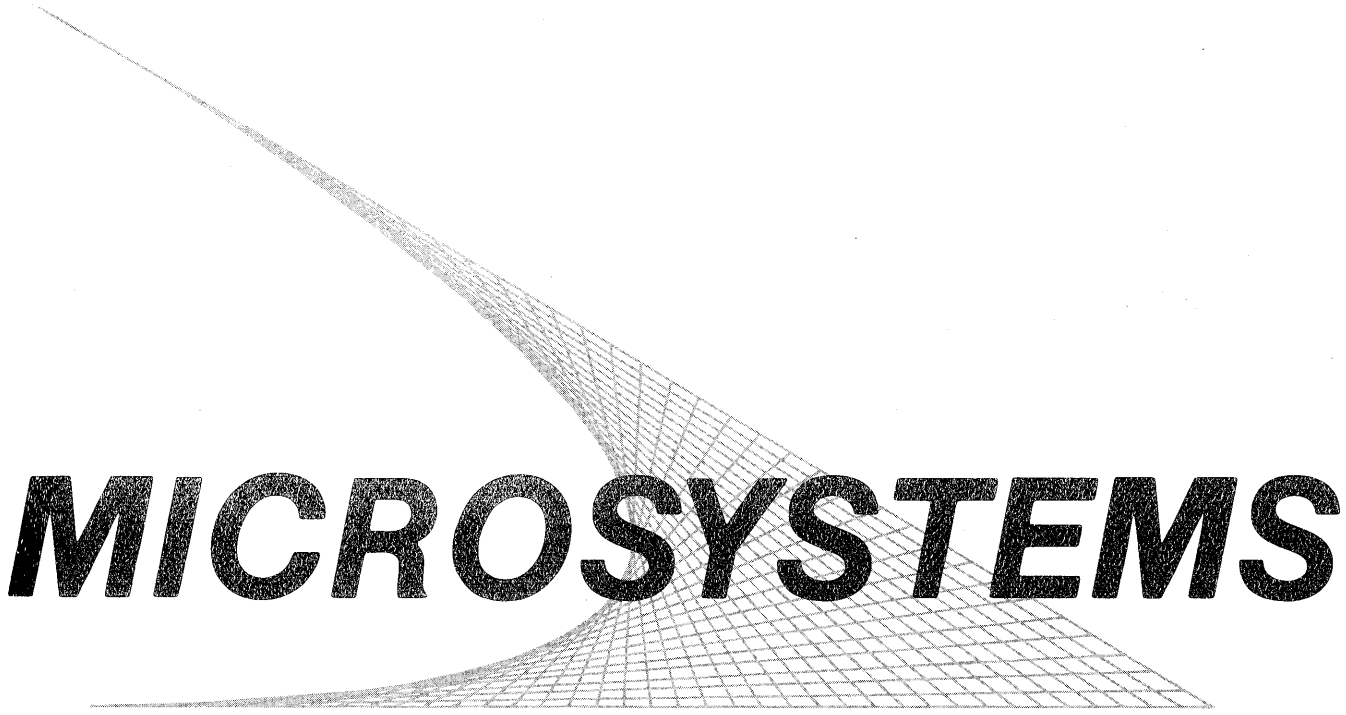
MOTOROLA

M68KMASM/D8

M68000 Family
Resident Structured Assembler
Reference Manual



MICROSYSTEMS

QUALITY • PEOPLE • PERFORMANCE

M68KMASM/D8

JULY 1984

M68000 FAMILY

RESIDENT STRUCTURED ASSEMBLER

REFERENCE MANUAL

This edition incorporates the information in any addendums to previous releases of this manual.

Eighth Edition

Seventh Edition July 1983

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

LIST OF TABLES

CHAPTER 1

GENERAL INFORMATION

1.1 SCOPE

The intent of this publication is to provide sufficient information to develop M68000 family assembly language programs, which may be run on MC68000-, MC68010-, or MC68020-based systems. The information herein pertains to the elements of the assembler. Detailed information pertaining to the MC68000 family of microprocessors is provided in the M68000 16/32-bit Microprocessor Programmer's Reference Manual. It is assumed that the designer has a complete understanding of the microprocessor architecture before attempting software development.

Chapters 1 through 4 contain the basic features of the assembler needed by the beginning assembly language programmer. Chapter 4 also provides instructions to invoke the assembler. Advanced topics, such as macro operations, conditional assembly, and structured syntax, are described in Chapters 5 through 8.

1.2 INTRODUCTION

The M68000 Family Resident Structured Assembler (referred to as the "assembler" throughout this manual) is used to translate M68000 family assembler source programs into MC68000/MC68010/MC68020 machine language. The assembler executes under VERSAdos on the EXORmacs Development System, the VERSAmodule 01, 02, or 03 Monoboard Microcomputer, the VMC 68/2 Microcomputer System, the VME/10 Microcomputer System, VMEmodule Monoboard Microcomputer (MVME101 or MVME110), or under SYSTEM V/68 on the EXORmacs Development System or the VME/10 Microcomputer System.

The assembler includes the following features:

- Absolute/relocatable code generation
- Complex expressions
- Symbol table listing
- Macros
- Conditional assembly
- Structured syntax
- Cross-reference
- IEEE Standard floating-point data types (MC68881 only)

1.3 M68000 FAMILY ASSEMBLY LANGUAGE

The symbolic language used to code source programs for processing by the assembler is called assembly language. This language is composed of the following symbolic elements:

a. Symbolic names or labels, which represent instruction, directive, and register mnemonics, as well as user-defined memory labels and macros.

b. Numbers, which may be represented in binary, octal, decimal, IEEE standard floating-point (MC68881 only), or Binary Coded Decimal (BCD) notation.

c. Arithmetic and logical operators, which are employed in complex expressions.

d. Special-purpose characters, which are used to denote certain operand syntax rules, macro functions, source line fields, and numeric bases.

1.3.1 Machine-Instruction Operation Codes

Appendix A summarizes that part of the assembly language that provides mnemonic machine-instruction operation codes for the MC68000, MC68010, MC68020, and MC68881 machine instructions.

1.3.2 Directives

The assembly language contains mnemonics for directives which specify auxiliary actions to be performed by the assembler. Directives are not always translated to machine language.

Assembler directives assist the programmer in controlling the assembler output, in defining data and symbols, and in allocating storage.

1.4 M68000 FAMILY RESIDENT STRUCTURED ASSEMBLER

The assembler translates source statements written in the assembly language into relocatable or absolute object code, assigns storage locations to instructions and data, and performs auxiliary assembler actions designated by the programmer. Object modules produced by the assembler are compatible with the M68000 family Linkage Editor or the SYSTEM V/68 PAL Linkage Editor, both also referred to as the "linkage editor" or "linker".

The assembler includes macro and conditional assembly capabilities, and implements certain "structured" programming control constructs. The assembler generates object code which may then be linked into a memory image format.

1.4.1 Assembler Purposes

The two basic purposes of the assembler are to:

- Provide the programmer with the means to translate source statements into object code -- that is, to the format required by the linkage editor.
- Provide a printed listing containing the source language input, assembler object code, and additional information (such as error codes, if any) useful to the programmer.

1.4.2 Assembler Processing

Assembly is a two-pass process. During the first pass, the assembler develops a symbol table, associating user-defined labels with values and addresses. During the second pass, the translation from source language to machine language takes place, using the symbol table developed during pass 1. In pass 2, as each source line is processed in turn, the assembler generates appropriate object code and the assembly listing.

1.4.3 Microprocessor Types

The assembler in its default mode provides assembly of instructions for the MC68000 processor. However, the assembly of MC68010, MC68020, and MC68881 instructions can be enabled either as a directive in the source text which precedes instruction mnemonics or from the command line (refer to paragraphs 3.5.2.10 and 4.2.1, respectively).

1.5 RELOCATION AND LINKAGE

"Relocation" refers to the process of binding a program to a set of memory locations at a time other than during the assembly process. For example, if subroutine "ABC" is to be used by many different programs, it is desirable to allow the subroutine to reside in any area of memory. One way of repositioning the subroutine in memory is to change the "ORG" directive operand field at the beginning of the subroutine, and then to reassemble the routine. A disadvantage of this method is the expense of reassembling ABC. An alternative to multiple assemblies is to assemble ABC once. Produced is an object module, which contains enough information, so that another program (the linkage editor) can easily assign a new set of memory locations to the module. This scheme offers these advantages: reassembly is not required; the object module is substantially smaller than the source program; relocation is faster than reassembly, and relocation can be handled by the linkage editor (rather than by editing the source program and changing the ORG directive).

In addition to program relocation, the linkage editor must also resolve inter-program references. For example, the other programs that are to use subroutine ABC must contain a jump-to-subroutine instruction to ABC. However, since ABC is not assembled at the same time as the calling program, the assembler cannot put the address of the subroutine into the operand field of the subroutine call. The linkage editor, however, will know where the calling program resides and, therefore, can resolve the reference to the call to ABC. This process of resolving inter-program references is called "linking". An example of linking two object modules is shown in Appendix D.

Program sections provide the basis of the relocation and linking scheme. Each of these sections may also have a variable number of named common sections associated with it, with each common section having a unique name. These relocatable sections are passed on to the linkage editor. From the different modules that are to be linked, the linkage editor collects all sections with the same number. Each of the 16 relocatable sections may contain data and/or code; in addition, named common sections may be defined within any relocatable section.

Absolute sections are unnumbered (and, therefore, unlimited in number); they are specified by the ORG directive.

1.6 LINKER RESTRICTIONS

Before developing relocatable assembly language modules, the user should become familiar with the capabilities and restrictions of the linkage process, as outlined in the M68000 Family Linkage Editor User's Manual or the SYSTEM V/68 PAL Linkage Editor User's Manual. It is important to keep in mind that the relocation features of the assembler are directly attributable to capabilities of the linkage editor, and that the linkage environment can be controlled through assembler directives. If the assembly language object program is to be linked with a Pascal object program, the user should be aware of Pascal's requirements before allocation.

The assembler will produce an object module compatible with the linkage editor. XDEF and XREF must be used to define entry points into the various modules and external symbols appearing in the module.

1.7 NOTATION

Commands and other input/output (I/O) are presented in this manual in a modified Backus-Naur Form (BNF) syntax. Certain symbols in the syntax, where noted, are used in the real I/O; however, others are meta-symbols whose usage is restricted to the syntactic structure. These meta-symbols and their meanings are as follows:

< >	The angular brackets enclose a symbol, known as a syntactic variable, that is replaced in a command line by one of a class of symbols it represents. In some cases, where noted, angular brackets are required characters.
|	This symbol indicates that a choice is to be made. One of several symbols, separated by this symbol, should be selected.
[ ]	Square brackets enclose a symbol that is optional. The enclosed symbol may occur zero or one time. In some cases, where noted, square brackets are required characters.
[ ]...	Square brackets followed by periods enclose a symbol that is optional/repetitive. The symbol may appear zero or more times.

Operator entries are to be followed by a carriage return.

1.8 RELATED PUBLICATIONS

The user should be familiar with the following Motorola publications, as appropriate to system type.

EXORmacs Development System Operations Manual (M68KMACS)

VME/10 Microcomputer System Overview Manual (M68KVSOM)

VMC 68/2 Series Microcomputer System Manual (MVMCSM)

VERSAdos to VME Hardware and Software Configuration User's Manual (MVMEVDOS)

VERSAdos to VMEmodules Hardware and Software Configuration Manual (MVMECNFG1)

M68000 16/32-Bit Microprocessor Programmer's Reference Manual (M68000UM)

M68000 Family Linkage Editor User's Manual (M68KLINK)

M68000 Family Resident Pascal User's Manual (M68KPASC)

VERSAdos Messages Reference Manual (M68KVMSG)

VERSAdos System Facilities Reference Manual (M68KVSF)

SYSTEM V/68 Error Message Manual (M68KUNMSG)

SYSTEM V/68 PAL Linkage Editor User's Manual (M68KUNLNK)

SYSTEM V/68 Pascal Compiler User's Manual (M68KUNPAS)

SYSTEM V/68 User's Manual (M68KUNUM)

CHAPTER 2

SOURCE PROGRAM CODING

2.1 INTRODUCTION

A source program is a sequence of source statements arranged in a logical way to perform a predetermined task. Each source statement occupies a line of printable text, where each line may be one of the following:

a. Comment
b. Executable instruction
c. Assembler directive
d. Macro invocation

NOTE

The MC68020 assemblers running under VERSAdos or SYSTEM V/68 and the MC68000/MC68010 assembler running under SYSTEM V/68 are case-insensitive to source input except as noted under the INCLUDE directive or for ASCII strings. All instruction examples in this manual are in uppercase letters, excluding explanations.

2.2 COMMENTS

Comments are strings, composed of any ASCII characters (refer to Appendix B), which are inserted into a program to identify or clarify the individual statements or program flow. Comments are included in the assembly listing but are ignored by the assembler.

A comment may be inserted in one of two ways:

a. At the beginning of a line, starting in column one, where an asterisk (*) is the first character in the line. The entire line is a comment, and an instruction or directive in this line will not be recognized.

b. Following the operation and operand fields of an assembler instruction or directive, where it is preceded by at least one space (refer to paragraph 2.4.4).

Examples:

```
* THIS ENTIRE LINE IS A COMMENT.

  BRA LAB2    THIS COMMENT FOLLOWS AN INSTRUCTION.
```

2.3 EXECUTABLE INSTRUCTION FORMAT

Assembly language programs are translated by the assembler into object code that may contain executable instructions, data structures, and relocation information. This translation process begins with symbolic assembly language source code, which employs reserved mnemonics, special symbols, and user-defined labels. M68000 family assembly language is line-oriented.

2.4 SOURCE LINE FORMAT

Each source statement has an overall format that is some combination of the following four fields:

a. label
b. operation
c. operand
d. comment

The statement lines in the source file must not be numbered. The assembler, however, prefixes each line in the listing file with a sequential number, up to four decimal digits.

The format of each line of source code is described in the following paragraphs.

2.4.1 Label Field

The label field is the first field in the source line. A label which begins in the first column of the line may be terminated by either a space or a colon. A label may be preceded by one or more spaces, provided it is then terminated by a colon. In either case, the colon is not a part of the label.

Labels are allowed on all instructions and assembler directives which define data structures. For such operations, the label is defined with a value equal to the location counter for the instruction or directive, including a designation for the program section in which the definition appears.

Labels are required on the assembler directives which define symbol values (SET, EQU, REG). For these directives, the label is defined with a value (and for SET and EQU, a program section designation) corresponding to the expression in the operand field.

Labels on MACRO definitions are saved as the mnemonic by which that macro is subsequently invoked. No memory address is associated with such labels. A label is also required on the IDNT directive. This label is passed on to the relocatable object module; it has no associated internal value.

No other directives allow labels.

Labels which are the only field in the source line, are defined equal to the current location counter value and program section.

2.4.2 Operation Field

The operation field follows the label field and is separated from it by at least one space. Entries in the field fall under one of the following categories:

a. Instruction mnemonics - which correspond to the M68000 family processor instruction set.

b. Directive mnemonics - pseudo-operation codes for controlling the assembly process.

c. Macro calls - invocations of previously-described macros.

The size of the data field affected by an instruction is determined by the data size code. Some instructions and directives can operate on more than one data size. For these operations, the data size code must be specified or a default size is assumed. The size code need not be specified if only one data size is permitted by the operation. The data size code is specified by appending a period (.) to the operation field, followed by B, W, L, S, D, X, or P where:

B = Byte (8-bit data)

W = Word (16-bit data)

L = Longword (32-bit data)

S = Byte (8-bit offset for certain branch instructions)

S = Single precision binary real (IEEE Standard, 32-bit: 8-bit exponent, 23-bit mantissa, 1-bit sign) (MC68881 only)

D = Double precision binary real (IEEE Standard, 64-bit: 11-bit exponent, 52-bit mantissa, 1-bit sign) (MC68881 only)

X = Extended precision binary real (96-bit: 15-bit exponent, 64-bit mantissa, 1-bit sign) (MC68881 only), (16-bits are reserved)

P = Packed Binary Coded Decimal (BCD) real string (96-bit: 3-decimal digit exponent and 17-decimal digit mantissa) (MC68881 only)

The data size code is not permitted, however, when the instruction or directive does not have a data size attribute.

Examples (legal):

LEA	2(A0),A1	Longword size is assumed (.B, .W not allowed); this instruction loads effective address pointed to by A0, +2 into A1.
ADD.B	ADDR,D0	This instruction adds byte whose address is ADDR to low order byte in D0.
ADD	D1,D2	This instruction adds low order word of D1 to low order word of D2. (W is the default size code.)
ADD.L	A3,D3	This instruction adds entire 32-bit (longword) contents of A3 to D3.

Example (illegal):

SUBA.B	#5,A1	Illegal size specification (.B not allowed on SUBA). This instruction attempts to subtract the value 5 from the low order byte of A1; byte operations on address registers are not allowed.

2.4.3 Operand Field

If present, the operand field follows the operation field and is separated from the operation field by at least one space. When two or more operand subfields appear within a statement, they must be separated by a comma but may not contain embedded spaces; e.g., D1, D2 is illegal. In an instruction like 'ADD D1,D2', the first subfield (D1) is generally applied to the second subfield (D2) and the results placed in the second subfield. Thus, the contents of D1 are added to the contents of D2; the result is saved in register D2. In the instruction 'MOVE D1,D2', the first subfield (D1) is the sending field; the second subfield (D2) is the receiving field. In other words, for most two-operand instructions, the general format 'opcode source,destination' applies.

2.4.4 Comment Field

The last field of a source statement is an optional comment field. This field is ignored by the assembler except for being included in the listing. The comment field is separated from the operand field (or the operation field, if there is no operand) by one or more spaces and may consist of any ASCII characters. This field is important in documenting the operation of a program.

2.5 ADDRESSING MODES

Effective address modes, combined with operation codes, define the particular function to be performed by a given instruction. Effective addresses and data organization are described in detail in Section 2, "Data Organization and Addressing Capabilities", of the M68000 16/32-Bit Microprocessor Programmer's Reference Manual.

References to data addresses may be odd only if a byte is referenced. Data references involving words or longwords must be even. Likewise, instructions must begin on an even byte boundary.

Individual bits within a byte (operand for memory destinations) or longwords (operands for Data register destinations) may be addressed with the bit manipulation instructions (paragraph 2.10.6). Bits for a byte are numbered 7 to 0, with 7 being the most significant bit position and 0 the least significant. Bits for a word are numbered 15 to 0, with 15 being the most significant bit and 0 the least significant. Bits for a longword are numbered from 31 to 0, with 31 being the most significant bit position and 0 the least significant bit position.

The code generated in the listing file for some addresses may be the same as the code generated for different expressions whenever externally referenced symbols are involved. The object file contains the correctly resolved addresses.

Following are definitions of the symbols used in Tables 2-1 and 2-2 and throughout the remainder of this section:

An	Address register number "n" (0-7).
ZAn	(MC68020 only) Suppressed address register number "n" (0-7) whose value is taken to be zero. Can be used in place of An if suppression is desired.
Dn	Data register number "n" (0-7).
Ri	(MC68020 only) Index register number "i"; may be any address (An) or data (Dn) register with optional ".W" or ".L" size designation (16 vs 32 bits). Scaling factor "scl" may also exist.

ZRi	(MC68020 only) Suppressed index register number "i" (0-7) whose value is taken to be zero. Can be used in place of Ri if suppression is desired.
scl	(MC68020 only) Scaling factor of 1, 2, 4, 8 optionally used in indexing modes. The default is 1.
PC	Program counter.
ZPC	(MC68020 only) Suppressed program counter whose value is taken to be zero. Can be used in place of PC if suppression is desired.
B,W,L	Byte, word, longword data sizes.
d(An)	Address register indirect with displacement (d).
d(An,Ri)	Address register indirect with index (Ri) plus displacement (d).
d(PC)	Program counter with displacement (d).
d(PC,Ri)	Program counter with index (Ri) plus displacement (d).
<absolute>	Absolute expression.
<simple>	Simple relocatable expression.
<complex>	Complex relocatable expression.
bd	(MC68020 only) Base displacement that is added before indirection occurs.
od	(MC68020 only) Outer displacement that is added after indirection occurs. Displacement size may be either word or longword.
<ea>	Effective address expression.
<iea>	Indirect effective address expression.
null	(MC68020 only) Null displacements imply that no extension word is present in the instruction for displacement.
Omitted values	(MC68020 only) Omitted registers take on suppressed register values (taken to be zero). Omitted displacements take on null values (taken to be zero).
Grouping characters	(MC68020 only) [ ] enclose an indirect expression and are required characters. ( ) enclose the entire <ea> expression and are required characters.
Order	(MC68020 only) Addressing arguments may occur in any order within the grouping characters. When two registers appear in an <ea> expression, if the leftmost could be either An or Ri, then a base register An is assumed for the leftmost, and the second is taken as an index register Ri.

Table 2-1 summarizes the addressing modes defined for the M68000 family, their invocations, and significant constraints.

TABLE 2-1. Address Modes

MODE	INVOCATION	COMMENTS
1) Register direct	An Dn	
2) Memory address		
a) Simple indirect	(An)	
b) Predecrement	-(An)	
c) Postincrement	(An)+	
d) Indirect with displacement (16-bit)	<absolute>(An) <complex>(An)	
e) Indirect with index (16- or 32-bit) plus displacement (8-bit)	<absolute>(An,Ri)	Due to linker constraints, any odd-addressed labels, used with externally defined labels, will generate a "break to odd address" error.
f) Indirect with preindexing plus base and outer displacements (MC68020 only)	([bd,An,Ri{*scl}],od)	
g) Indirect with postindexing plus base and outer displacements (MC68020 only)	([bd,An],Ri{*scl},od)	
h) Direct with indexing plus base displacement (MC68020 only)	(bd,An,Ri{*scl})	
3) Special address		
a) PC with displacement (16-bit)	<simple>	Expression is an address (not a displacement) which must be backward, within current relocatable section.
	<absolute>(PC) <simple>(PC) <complex>(PC)	Forced PC-relative. Must fit within 16-bit signed field; resolved at assembly or link time.

TABLE 2-1. Address Modes (cont'd)

MODE	INVOCATION	COMMENTS
	<absolute>(PC) <simple>(PC) <complex>(PC)	Forced PC-relative. Must fit within 16-bit signed field; resolved at assembly or link time.
b) PC with index (16- or 32-bit) plus displacement (8-bit)	<simple>(Ri)	Expression is an address which must be backward, within current relocatable section. Also, due to linker constraints, any odd-addressed labels, used with externally defined labels, will generate a "break to odd address" error.
	<absolute>(PC,Ri) <simple>(PC,Ri)	Forced PC-relative; expression must be within current program section.
c) PC with preindexing plus base and outer displacements (MC68020 only)	([bd,PC,Ri{*scl}],od)	
d) PC with postindexing plus base and outer displacements (MC68020 only)	([bd,PC],Ri{*scl}],od)	
e) PC direct with indexing plus base (MC68020 only)	(bd,PC,Ri{*scl})	
f) Absolute (16- or 32-bit)	<absolute> <complex> <simple>	Expression must be forward reference or not in current program section.
g) Immediate (8-, 16-, or 32-bit)	#<absolute> #<simple> #<complex>	Due to linker constraints, any odd-addressed labels, used with externally defined labels, will generate a "break to odd address" error.

TABLE 2-1. Address Modes (cont'd)

MODE	INVOCATION	COMMENTS
4) Implicit PC reference		Invoked by conditional branch (Bcc) or DBcc instruction; the effective address is a displacement from the PC; the displacement is either 8, 16, or 32 bits (32 on MC68020 only), depending on OPT BRS, OPT BRB, OPT BRW, and OPT BRL, and whether these options are overridden on the current instruction (see paragraph 2.6). Also, due to linker constraints, any odd-addressed labels, used with externally defined labels, will generate a "break to odd address" error.

Table 2-2 provides a cross reference of operand formats and addressing modes. Given an operand of the format shown in the first column, the other columns show which addressing mode is indicated, depending on whether the expression is absolute, simple relocatable, or complex relocatable.

TABLE 2-2. Cross-Reference: Effective Addressing Mode, Given Operand Format and <expr> Type

	EFFECTIVE ADDRESSING MODE		
OPERAND FORMAT	ABSOLUTE <expr>	SIMPLE RELOCATABLE <expr>	COMPLEX RELOCATABLE <expr>
<expr>(An)	d(An)	d(An)	d(An)
<expr>(Dn)	invalid	d(PC,Dn) *	invalid
<expr>(An,Ri)	d(An,Ri)	invalid	invalid
<expr>	absolute (W,L)	d(PC) or absolute (W,L)	absolute (W,L)
<expr>(PC)	d(PC)	d(PC)	d(PC)
<expr>(PC,Ri)	d(PC,Ri) *	d(PC,Ri) *	invalid
#<expr>	immediate(B,W,L)	immediate (W,L)	immediate (W,L)

* Must be within current program section.

2.5.1 Register Direct Modes

These effective addressing modes specify that the operand is in one of the 16 multifunction registers (eight data and eight address registers). The operation is performed directly on the actual contents of the register.

Notations: An, Dn where n is between 0 and 7

Examples:			
	CLR.L	D1	Clear all 32 bits of D1.
	ADD	A1,A2	Add low order word of A1 to low order word of A2.

2.5.2 Memory Address

The following effective addressing modes specify that the operand is in memory and provide the specific address of the operand.

2.5.2.1 Address Register Indirect. The address of the operand is in the address register specified by the register field.

Notation: (An)

Examples:			
	MOVE	#5,(A5)	Move value 5 to word whose address is contained in A5.
	SUB.L	(A1),D0	Subtract from D0 the value in the long word whose address is contained in A1.

2.5.2.2 Address Register Indirect with Postincrement. The address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two, or four, depending upon whether the size of the operand is byte (.B), word (.W), or long (.L).

Notation: (An)+

Examples:			
	MOVE.B	(A2)+,D2	Move byte whose address is in A2 to low order byte of D2; increment A2 by 1.
	MOVE.L	(A4)+,D3	Move longword whose address is in A4 to D3; increment A4 by 4.

2.5.2.3 Address Register Indirect with Predecrement. The address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or four, depending upon whether the operand size is byte (.B), word (.W), or long (.L).

Notation: -(An)

Examples:

```
CLR     -(A2)       Subtract 2 from A2; clear word whose
                    address is now in A2.

CMP.L   -(A0),D0    Subtract 4 from A0; compare longword
                    whose address is now in A0 with contents
                    of D0.
```

2.5.2.4 Address Register Indirect with Displacement. The address of the operand is the sum of the address in the address register and the sign-extended displacement.

Notation: d(An)

Examples:

```
AVAL    EQU     5           AVAL is equated to 5 (for use in next
                            instruction).

        CLR.B   AVAL(A0)    Clear byte whose address is given by
                            adding value of AVAL (=5) to contents
                            of A0.

        MOVE    #2,10(A2)   Move value 2 to word whose address is
                            given by adding 10 to contents of A2.
```

2.5.2.5 Address Register Indirect with Index. The address of the operand is the sum of the address in the address register, the sign-extended displacement, and the contents of the index (A or D) register.

Notations:

```
d(An,Ri)      Specifies low order word of index register.
d(An,Ri.W)
d(An,Ri.L)    Specifies entire contents of index register.
```

Examples:

```
ADD     AVAL(A1,D2),D5      Add to low order word of D5 the word
                            whose address is given by addition of
                            contents of A1, the low order word of
                            index register (D2), and the displacement
                            (AVAL).

MOVE.L  D5,$20(A2,A3.L)     Move entire contents of D5 to longword
                            whose address is given by addition of
                            contents of A2, contents of entire index
                            register (A3), and the displacement
                            ($20).
```

2.5.2.6 Address Register Indirect with Preindexing Plus Base and Outer Displacements. (MC68020 only) The address of the operand is the sum of the <iea> and a sign-extended outer displacement value od. <iea> is the sum of the contents of the address register An (or ZAn), the base displacement bd, and the contents of the index register Ri (or ZRi). Therefore,

```
bd + (An) + Ri  --->  <iea>
 (<iea>) + od   --->  <operand>
```

Notation:

([bd,An,Ri{*scl}],od) or ([bd,An,Ri.W{*scl}],od)	Specifies low-order word of index register Ri.
([bd,An,Ri.L{*scl}],od)	Specifies entire contents of index register Ri.

Examples:

ADD ([BASE,A1,D2],AVAL),D5	The sum of the value of BASE, the contents of base register A1, and the contents of the low-order word of index register D2 points to <iea>. The contents of the resultant address <iea> added to the value of AVAL give the <ea> of the operand to be added to the contents of D5.
ADD ([2,A1,A2],4),D5	In this example, the assembler selects the leftmost A register (A1) to be the base register.

2.5.2.7 Address Register Indirect with Postindexing Plus Base and Outer Displacements. (MC68020 only) The address of the operand is the sum of the <iea>, the contents of the index register Ri (or ZRi), and the outer displacement value od. <iea> is the sum of the base displacement bd and the contents of the base register An (or ZAn). Therefore,

```
          bd + (An)   --->  <iea>
(<iea>) + od + Ri     --->  <operand>
```

Notation: ([bd,An],od,Ri{*scl}) or
([bd,An],od,Ri.W{*scl}) Specifies low-order word of index register Ri.

([bd,An],od,Ri.L{*scl}) Specifies entire contents of index register Ri.

Example: `ADD ([BASE,A1],AVAL,D2),D5` The sum of the value of BASE and the contents of base register A1 points to <iea>. The contents of the resultant address <iea> added to the value of AVAL and the low-order word of index register D2 points to the address of the operand to be added to the contents of D5.

2.5.2.8 Address Register Direct with Indexing Plus Base Displacement. (MC68020 only) The address of the operand is the sum of the 8-bit base displacement bd, the contents of the base register An, and the contents of the index register Ri. Therefore,

bd + (An) + (Ri) ---> <operand>

Notation: (bd,An,Ri{*scl}) or (bd,An,Ri.W{*scl}) Specifies low-order word of index register Ri.

(bd,An,Ri.L{*scl}) Specifies entire contents of index register Ri.

Example:

```
ADD    (BASE,A1,D2),D5
```

The sum of the value of BASE, the contents of base register A1, and the low-order word of index register D2 points to the address of the operand to be added to the contents of D5.

```
ADD    (BASE,A1,A2),D5
```

In this example, A1 is the base register because it is the leftmost candidate for base register. A2 is interpreted as being an index register.

2.5.3 Special Address Modes

Special address modes use the effective address register field to specify the special addressing mode instead of a register number. Table 2-3 provides the ranges for absolute short and long addresses.

TABLE 2-3. Special Address Ranges

32-BIT ADDRESS	16-BIT REPRESENTATION OF 32-BIT ADDRESS
00000000 . . 00007FFF	0000 . . 7FFF Absolute short
00008000 . . FFFF7FFF	(No representation in 16 bits; must be absolute long)
FFFF8000 . . FFFFFFFF	8000 . . FFFF Absolute short

2.5.3.1 Absolute Short Address. The 16-bit address of the operand is sign extended before it is used. Therefore, the useful address range is 0 through $7FFF and $FFFF8000 through $FFFFFFFF.

Notation: XXX

Example: JMP $400 Jump to hex address 400

(MC68020 only) An absolute short address can be forced by using the notation:

(XXX).W

2.5.3.2 Absolute Long Address - The address of the operand is the 32-bit value specified.

Notation: XXX

Example: JMP $12000 Jump to hex address 12000

(MC68020 only) An absolute long address can be forced by using the notation:

(XXX).L

2.5.3.3 Program Counter with Displacement. The address of the operand is the sum of the address in the program counter and the sign-extended displacement integer. The assembler calculates this sign-extended displacement by subtracting the address of displacement word from the value of the operand field.

Notation:	<expression>(PC)		Forced program counter-relative. Note that <expression> is interpreted as a program address rather than a displacement.
Example:	JMP	TAG(PC)	Force the jump to address TAG to be program counter-relative.

2.5.3.4 Program Counter with Index. The address is the sum of the address in the program counter, the sign-extended displacement value, and the contents of the index (A or D) register.

Notations:	<expression>(Ri.W)		Specifies low order word of index register. .W is optional (default).
	<expression>(Ri.L)		Specifies entire contents of index register.
	<expression>(PC,Ri)		Forced program counter-relative. Ri.W or Ri.L legal. NOTE: <expression> is interpreted as a program address rather than a displacement.
Examples:	MOVE	T(D2),TABLE	Moves word at location (T plus contents of D2) to word location defined by TABLE. T must be a relocatable symbol.
	JMP	TABLE(A2.W)	Transfers control to location defined by TABLE plus the lower 16-bit content of A2 with sign extension. TABLE must be a relocatable symbol.
	JMP	TAG(PC,A2.W)	Forces evaluation of TAG to be program counter-relative with index.

2.5.3.5 Program Counter with Preindexing Plus Base and Outer Displacements. (MC68020 only) The address of the operand is the sum of the <iea> and a sign-extended outer displacement value od. <iea> is the sum of the contents of the Program Counter PC (or ZPC), the base displacement bd, and the contents of the index register Ri (or ZRi). Therefore

bd + (PC) + Ri ---> <iea>
(<iea>) + od ---> <operand>

NOTE

Whenever ZPC is used, bd is not offset by the current PC value. od is never offset by the PC value.

Notation: ([bd,PC,Ri{*scl}],od) or
([bd,PC,Ri.W{*scl}],od) Specifies low-order word of index register Ri.

([bd,PC,Ri.L{*scl}],od) Specifies entire contents of index register Ri.

Examples: ADD ([BASE,PC,A2],AVAL),D5 The sum of the value of BASE, the contents of the program counter PC, and the contents of the low-order word of index register 2 points to <iea>. The contents of the resultant address <iea> added to the value of AVAL give the <ea> of the operand to be added to the contents of D5.

ADD ([A2,PC,BASE],AVAL),D5 This example is equivalent to the example above because ordering of operands is not required.

2.5.3.6 Program Counter with Postindexing Plus Base and Outer Displacements. (MC68020 only) The address of the operand is the sum of the <iea>, the contents of the index register Ri (or ZRi), and the outer displacement value od. <iea> is the sum of the base displacement bd and the contents of the program counter PC (or ZPC). Therefore,

```
          bd + (PC)  --->  <iea>
(<iea>) + od + Ri    --->  (operand>
```

Notes: Whenever ZPC is used, bd is not offset by the current PC value. od is never offset by the PC value.

Notation:

([bd,PC],od,Ri{*scl}) or ([bd,PC],od,Ri.W{*scl})	Specifies low-order word of index register Ri.
([bd,PC],od,Ri.L{*scl})	Specifies entire contents of the index register Ri.

Example: `ADD ([BASE,PC],AVAL,D2),D5`

The sum of the value of BASE and the contents of program counter PC points to <iea>. The contents of the resultant address <iea> added to the value of AVAL and the low-order word of index register D2 points to the address of the operand to be added to the contents of D5.

2.5.3.7 Program Counter Direct with Indexing Plus Base Displacement. (MC68020 only) The address of the operand is the sum of the sign-extended 8-bit base displacement bd, the contents of the program counter PC, and the contents of the index register Ri. Therefore,

bd + (PC) + (Ri) ---> <operand>

Notation: (bd,PC,Ri{*scl}) or (bd,PC,Ri.W{*scl}) — Specifies low-order word of index register Ri.

(bd,PC,Ri.L{*scl}) — Specifies entire contents of the index register Ri.

Example: ADD (BASE,PC,D2),D5 — The sum of the value of BASE, the contents of program counter PC, and the contents of the low-order word of D2 points to the address of the operand to be added to the contents of D5.

2.5.3.8 Immediate Data. An absolute number may be specified as an operand by immediately preceding a number or expression with an immediate character. The immediate character (#) is used to designate an absolute number other than a displacement or an absolute address.

Notation: #XXX

Examples:

MOVE	#1,D0	Move value 1 to low order word of D0.
SUB.L	#1,D0	Subtract value 1 from the entire contents of D0.

2.6 NOTES ON MC68020 ADDRESSING MODES

There are new features in the MC68020 addressing modes. These features are discussed in the following paragraphs and are summarized in Table 2-4.

2.6.1 Address Register Addressing Modes

One of the main changes to the addressing modes in the MC68020 is in the mode 6 <ea> expressions. Some source code variations of the new mode 6 <ea> expressions are redundant with the MC68000 modes 2 and 5 (i.e., the final effective address is the same). When a redundant mode occurs, the mode 2 and 5 forms are selected by the assembler because they are more efficient. For example, when the assembler sees the following form:

```
(An)
```

it will generate a mode 2 addressing mode. Furthermore, the assembler will generate a mode 5 address when seeing the following two forms:

```
bd(An)      or the new syntax form
(bd,An)     when bd fits in 16 bits or less
```

The programmer can generate the redundant mode 6 instructions by using the suppressed registers. In the bd (An) form, bd must fit in 16 bits or less or an error (250) is generated. The (bd, An) form supports a bd up to 32 bits.

It is important to note that the assembler still recognizes the current 68000 syntax for mode 6 addresses. These two forms are:

```
(An,Ri)
bd(An,Ri)  or the new notation (bd,An,Ri)
```

They generate mode 6 addresses. However, the object code for the form written in new notation is different if a scaling factor other than one is present or bd cannot be represented in 8 bits or less.

Where new addressing modes are redundant with old addressing modes or with other new addressing modes, the assembler defaults to the more efficient addressing mode. However, less efficient forms can still be generated.

In general, old addressing modes are more efficient than the new modes. Within the new modes, pre-indexed indirect is more efficient than post-indexed indirect, and use of the index register is more efficient than use of the base address register for indirect modes.

Efficiency as used in this document refers to execution time. In most cases, the fastest variation is also the shortest one.

In the variation (bd,Ai*scl), the form (bd,Ai) is accepted. However, if the base displacement is less than or equal to 16 bits, the assembler automatically selects mode 5.

2.6.2 Program Counter Relative Addressing Modes

Another major change to the addressing modes in the MC68020 is in the mode 73 forms. Some of the new mode 73 addressing modes are redundant with the MC68000 mode 72. When a redundant mode occurs, the mode 72 form is used since it is more efficient. For example, when the assembler sees

bd(PC) or the new syntax form

(bd,PC) when bd fits in 16 bits or less

it generates a mode 72 address. The programmer can generate the redundant mode 73 instructions by using suppressed registers.

It is also important to note that the assembler recognizes the current 68000 syntax for mode 73 addresses. These forms are

(PC,Ri) or (PC) or bd(PC,Ri)

All mode 73 <ea> expressions require 'PC' or 'ZPC' as part of the expression to distinguish them from their address register counterparts. (All mode 72 and 73 references are to program space and all mode 2, 5, and 6 references are to data space.)

Where new addressing modes are redundant with old addressing modes or with other new addressing modes, the assembler defaults to the more efficient addressing mode. However, less efficient forms can still be generated.

When the program counter is suppressed (ZPC), the displacement is assumed to be absolute and hence is not offset from the current PC value.

2.6.3 Using Suppressed Registers to Force Redundant Addressing Modes

Register mnemonics ZPC, ZA0-ZA7 and ZD0-ZD7 imply registers whose values are always taken to be zero. These symbols may be used to specify any allowable register while at the same time suppressing that register during <ea> calculations. These symbols are included for diagnostic purposes so that every field of the object code instruction can be specified. It also indicates whether PC or An is being suppressed, and this determines whether the <ea> is in instruction space or data space. By default, An is taken to be the suppressed register if no register is specified. 'ZPC' must appear in the <ea> expression to force PC-relative addressing with PC suppressed.

Where an <ea> expression would normally default to a current 68000 addressing mode, the equivalent <ea> may be forced in mode 6 or 73 by including 'ZRi' within the <ea> expression. This is because the assembler always selects the most efficient addressing mode unless another equivalent mode is forced.

'ZRi' following the closing square bracket (i.e., '([<ea>],ZRi)') forces post-indexed indirect modes where the index register has been suppressed.

Registers can be suppressed only in the address register indirect and the Program Counter indirect modes.

2.6.4 Addressing Summary

Table 2-4 summarizes much of the information presented in the preceding paragraphs:

TABLE 2-4. Addressing Summary

DEFAULT ADDRESSING MODE		FUNCTIONALLY EQUIVALENT FORCED ADDRESSING MODES	
SYNTAX	MODE	SYNTAX	MODE
0	Mode 70	(ZRi)	Mode 6n w/null bd
0	Mode 70	((0).W,ZRi)	Mode 6n w/16-bit bd=0
0	Mode 70	((0).L,ZRi)	Mode 6n w/32-bit bd=0
bd	Mode 70	(bd,ZRi)	Mode 6n
(An)	Mode 2n	(An,ZRi)	Mode 6n
bd(An)	Mode 5n	(bd,An,ZRi)	Mode 6n
none		(ZPC)	Mode 73 w/null bd
none		((0).W,ZPC)	Mode 73 w/16-bit bd=0
none		((0).L,ZPC)	Mode 73 w/32-bit bd=0
none		(bd,ZPC)	Mode 73
([])	Mode 6n w/null bd	([(0).W])	Mode 6n w/16-bit bd=0
([])	Mode 6n w/null bd	([(0).L])	Mode 6n w/32-bit bd=0
([])	Mode 6n w/pre-ind.	([],ZRi)	Mode 6n w/post-ind.
([])	Mode 6n w/supp. An	([ZPC])	Mode 73 w/supp. PC

NOTE: "n" mode numbers refer to the base address register.

2.7 NOTES ON ADDRESSING OPTIONS

By default, the assembler resolves all forward references by using the longer form of the effective address in the operand reference. The programmer may override this default by specifying OPT FRS, which designates that forward absolute references should be short, or OPT BRB (or BRS), designating that forward relative branches should use the shorter (8-bit) displacement format. For the MC68020, OPT BRW can be used to force 16-bit (rather than 32-bit) displacements on forward branches.

On an instruction which does not allow a size code, the current forward reference default format may be overridden (for that instruction only) by appending .S (short) or .L (long) to the instruction mnemonic. A similar override may be performed in the structured syntax control directives via the extent codes (see paragraph 6.3 for further explanation). No override is possible on instructions with size code specification. Notably, this override procedure is possible on the JMP and JSR instructions.

The shorter form of the effective address for relative branch instructions is an 8-bit displacement; the longer format is a 16-bit displacement. For absolute jumps, the shorter effective address is the 16-bit absolute short; the longer format is the 32-bit absolute long mode. In the case of forward references in either relative branches or absolute jumps, if the shorter format is directed and the longer format is later found necessary when the reference is resolved, an error will occur.

References to symbols already defined, whether absolute or relative, are resolved by the assembler into the appropriate effective address, unless .S or .L is forced on the instruction.

A short form may be forced by following the instruction mnemonic with .S.

Example:

```
BEQ.S   LOOP1       If condition code 'EQ' (equal) is true, then branch to
                    LOOP1 (using the short form of the instruction).
```

In this case, the instruction size is forced to one word. An error will be printed if the operand field is not in the range of an 8-bit displacement.

Since 8-bit value fields are not relocated, a Bcc.S instruction, which branches to an XREF or other expression-required location, is not allowed. Such an instruction format results in an assembler error. A relative branch to a symbol known to be an XREF, or in a different section than the instruction, employs the longer (16-bit) displacement, with resolution done by the linkage editor.

Default actions of the assembler have been chosen to minimize two common address mode errors:

a. Displacement range violations

   Relative branch instructions (Bcc, BRA, BSR) allow either 8-bit or 16-bit displacements from the PC. On forward references in such instructions, the default action is to assume the 16-bit displacement (OPT BRW), which also allows resolution by the linkage editor, should that prove necessary.

b. Inappropriate absolute short address

   Absolute addresses may be short (16-bit) or long (32-bit). On forward references with absolute effective address, the default action is to assume the long format (OPT FRL). The long form is also assumed on references to another section (unless it is a SECTION.S), so that resolution by the linkage editor is assured.

Default conditions have been chosen to prevent errors by using addressing formats which ensure address resolution in the broadest range of conditions, at the expense of code efficiency. Each default may be overridden to improve efficiency or to create position independent code. Also, the current address size defaults (options FRL and FRS) may be overridden in certain cases on specific instructions which do not allow size codes by appending .S or .L, as in JMP.S and JMP.L (JMP and JSR only).

The previous discussion assumed relative branches could not be 32 bits. This is not the case when using the MC68020.

The resolution of operands into effective address modes (ignoring base register addressing) is summarized in the Tables 2-5 through 2-10.

TABLE 2-5. Operand Resolution

	INSTRUCTION FOLLOWS	
OPERAND TYPE	SECTION	ORG
Known location (backward in pass 1)	See Table 2-6	See Table 2-7
Unknown location (forward)	See Table 2-8	See Table 2-8
External reference	See Table 2-9	See Table 2-10

TABLE 2-6. Known* Location of Operand & Instruction Follows SECTION

OPERAND REFERENCE	OPTION IN EFFECT WHEN INSTRUCTION OCCURRED	EFFECTIVE ADDRESS MODE
Simple relocation	PCS	PC relative (resolved by linkage editor if operand & instructions are in different SECTIONs) IF displacement > 16-bit THEN error
	NOPCS (default)	IF operand and instruction in same SECTION and displacement <= 16-bit THEN PC relative ELSE IF operand defined in SECTION.S THEN absolute short ELSE absolute long (resolved by linkage editor)
Complex relocation	(Any)	Absolute long
Absolute (ORG)	(Any)	Absolute short or absolute long depending on the value of the operand

* Label defined before instruction which references it (in pass 1).

TABLE 2-7. Known* Location of Operand & Instruction Follows ORG

OPERAND REFERENCE	OPTION IN EFFECT WHEN INSTRUCTION OCCURRED	EFFECTIVE ADDRESS MODE
Simple relocation	(Any)	IF operand defined in SECTION.S THEN absolute short ELSE absolute long (resolved by linkage editor)
Complex relocation	(Any)	Absolute long
Absolute (ORG)	PCO	IF displacement <= 16-bit THEN PC relative ELSE absolute short or absolute long depending on value of operand
	NOPCO (default)	Absolute short or absolute long depending on the value of the operand

* Label defined before instruction which references it (in pass 1).

TABLE 2-8. Unknown** Location of Operand & Instruction Follows SECTION or ORG

OPERAND REFERENCE	OPTION IN EFFECT WHEN INSTRUCTION OCCURRED	EFFECTIVE ADDRESS MODE
(All)	FRS	Absolute short (resolved by linkage editor)
	FRL (default)	Absolute long (resolved by linkage editor)

** Label undefined at time of reference (error at pass 2).

TABLE 2-9. External Reference & Instruction Follows SECTION

OPERAND REFERENCE	OPTION IN EFFECT WHEN INSTRUCTION OCCURRED	EFFECTIVE ADDRESS MODE
XREF with SECTION designation	PCS	PC relative (resolved by linkage editor)
Example: XREF 2:L1	NOPCS (default)	IF operand defined in SECTION.S or XREF.S THEN absolute short ELSE absolute long (resolved by linkage editor)
XREF without SECTION designation	(Any)	IF operand defined with XREF.S THEN absolute short ELSE (see below) (resolved by linkage editor)
Example: XREF L1	FRS	Absolute short (resolved by linkage editor)
	FRL (default)	Absolute long (resolved by linkage editor)

TABLE 2-10. External Reference & Instruction Follows ORG

OPERAND REFERENCE	OPTION IN EFFECT WHEN INSTRUCTION OCCURRED	EFFECTIVE ADDRESS MODE
XREF with SECTION designation Example: XREF 2:L1	(Any)	IF operand defined in SECTION.S or XREF.S THEN absolute short ELSE absolute long (resolved by linkage editor)
XREF without SECTION designation Example: XREF L1	(Any)	IF operand defined with XREF.S THEN absolute short ELSE (see below) (resolved by linkage editor)
	FRS	Absolute short (resolved by linkage editor)
	FRL (default)	Absolute long (resolved by linkage editor)

2.8 SYMBOLS AND EXPRESSIONS

2.8.1 Symbols

Symbols recognized by the assembler consist of one or more valid characters (refer to Appendix B), of which the first eight are significant. The first character must be an uppercase letter (A-Z) or a period (.). Each remaining character may be an uppercase letter, a digit (0-9), a dollar sign ($), a period (.), or an underscore (_). Lowercase letters can also be used at times (refer to note in paragraph 2.1).

Numbers recognized by the assembler include decimal, hexadecimal, octal, and binary values. Decimal numbers (the default) are specified by a string of decimal digits (0-9); hexadecimal numbers are specified by a dollar sign ($) followed by a string of hexadecimal digits (0-9, A-F); octal numbers are specified by the commercial "at" sign (@) followed by a string of octal digits (0-7); binary numbers are specified by a percent sign (%) followed by a string of binary digits (0-1).

Examples:

Decimal - A string of decimal digits

Example: 12345

Hexadecimal - A dollar sign ($) followed by a string of hexadecimal digits

Example: $12345

Octal - An "at" sign (@) followed by a string of octal digits

Example: @12345

Binary - A percent sign (%) followed by a string of binary digits

Example: %10111

(MC68881 only) IEEE standard floating-point numbers can be specified by an optionally signed, fraction string of up to 17 decimal digits (0-9) containing a required decimal point, the constant "E", an optional sign (+ or -), and an exponent up to 3 decimal digits. The exponent section "E<sign>yyy" is optional; underscores can occur for readability.

Floating-point numbers can also be specified explicitly as a series of hexadecimal digits preceded by a colon (:). This floating-point hex format can be used to exactly represent the mantissa, exponent, and sign bit for a given floating-point number.

Examples:

Floating-point - sx.xxxxxxxxxxxxxxxxEsyyy (maximum size)

where: s is an optional sign;
x and y are decimal digits

Example: 1234.56E-33

Floating-point hex - :xxxxx...

where: xxxxx... is a sequence of hex digits
(up to 8 digits for .S precision, up to 16 for .D,
and up to 24 for .X or .P)

One or more ASCII characters enclosed by apostrophes (') constitute an ASCII string. ASCII strings are left-justified and zero-filled (if necessary), whether stored or used as immediate operands. This left justification will be to a word boundary if one or two characters are specified, or to a longword boundary if the string contains more than two characters. (In order to specify an apostrophe within a literal or string, two successive apostrophes must appear where the single apostrophe is intended to appear.)

```
Examples:  DC.L    'ABCD'
           DC.L    '''79'
           DC.W    '*'
           DC.L    'I''M'
```

2.8.2 Symbol Definition Classes

Symbols may be differentiated by usage into two general classes. Class 1 symbols are used in the operation field of the instruction (refer to paragraph 2.4 for field definitions); Class 2 symbols occur in the label and operand fields of the instruction. Assembler directives, instruction mnemonics, and macro names comprise Class 1 symbols; user-defined labels and register mnemonics are included in Class 2 symbols.

A Class 1 symbol may be redefined and used independently as a Class 2 symbol, and vice versa. As long as each symbol is used correctly, no conflict will result from the existence of two symbols of different classes with the same name. For example, the following is a legal instruction sequence:

```
           ADD   D1,ADD
           .
           .
           .
ADD        DS    2
```

By its usage as a Class 1 symbol, the first "ADD" is recognized as an instruction mnemonic; likewise, the second ADD is recognized as a Class 2 symbol identifying a reserved storage area. The assembler differentiates a Class 1 symbol from a Class 2 symbol with the same name, thereby allowing two symbol table entries with the same name but different class.

Macro labels are a special case because the same symbol will appear as the label (Class 2) in the MACRO definition and, subsequently, as an operation code mnemonic (Class 1) in invocation of that same macro. Macro labels are defined to be Class 1 symbols; their presence in the label field of a MACRO directive is ignored as a Class 2 symbol. Therefore, macro names may be redefined as Class 2 symbols without conflict.

A symbol may not be redefined within the same class. For example, ADD (reserved Class 1 symbol) may not be redefined as a macro label (also Class 1), nor may "A5" (reserved Class 2 symbol) be redefined as a statement or storage location label (also Class 2). A reserved symbol may be used only within its own class.

2.8.3 User-Defined Labels

Labels are defined by the user to identify memory locations in program or data areas of the assembly module. Each label has two attributes: the program section in which the memory location resides, and the offset from the beginning of that program section.

Labels may be defined to have an absolute or relocatable value, depending upon the program section in which the labeled memory location is found. If the memory location is within a relocatable section (defined through the SECTION directive), then the label has a relocatable value relative to that program section. If the memory location is not contained within a relocatable section (for example, the location follows an ORG directive), then the label has an absolute value.

Labels may be defined in the label field of an executable instruction or a data definition directive source line. It is also possible to SET or EQU a label to either an absolute or a relocatable value.

2.8.4 Integer Expressions

Expressions are composed of one or more symbols, which may be combined with unary or binary operations. Legal symbols in expressions include:

a. User-defined labels and their associated absolute or relocatable values.

b. Numbers and their absolute values.

c. The special symbol "*" always identifies the value of the program counter at the beginning of the DC directive, even when multiple arguments are specified (e.g., DC.B 1,2,3,*-3). The program counter may be either absolute or relocatable.

Subexpressions which involve relocatable symbols may employ only the "+" and "-" operators. It is possible for a subexpression involving the difference between two relocatable symbols to evaluate to an absolute value. For example, let R1 represent a memory location at OFFSET1 bytes beyond the start of section S1, and let R2 represent a memory location at OFFSET2 bytes beyond the start of section S2 -- that is,

```
R1 = OFFSET1 + <start of S1>
R2 = OFFSET2 + <start of S2>
```

The difference between R1 and R2 may then be

```
R1-R2 = OFFSET1-OFFSET2 + <start of S1> - <start of S2>
```

If sections S1 and S2 are the same, then

```
R1-R2 = OFFSET1-OFFSET2
```

which is a constant, absolute (non-relocatable) value. Of course, if sections S1 and S2 are distinct, the expression remains a complex, relocatable expression.

When an expression has been fully evaluated by the assembler, it may be categorized as one of three types of expressions:

a. Absolute expression - The expression has reduced to an absolute value which is independent of the start address of any relocatable section.

b. Simple relocatable expression - The expression has reduced to an absolute offset from the start of a single relocatable section.

c. Complex relocatable expression - The expression has reduced to a constant, absolute offset in conjunction with either of the following relocatable terms:

    1. A single, negated start address of a relocatable section.

    2. References to the start addresses of two or more relocatable sections; these references may be additions to or subtractions from the constant offset value.

NOTE

Complex relocatable expressions, such as an absolute symbol minus a relocatable symbol, are illegal in ORG, OFFSET, EQU, DCB, DS, COMLINE, and SET directives.

By themselves, all user-defined labels on memory locations are either absolute or simple relocatable expressions. This includes XREF labels, which are assumed to be absolute symbols unless their program section is specified. Complex relocatable expressions may arise only from the addition or subtraction of two relocatable expressions.

The following are examples of each type of expression.

```
          ORG       $1000
ARRAY     DS        $20             "ARRAY" is absolute
ENDARRAY  EQU       *-2             "ENDARRAY" is absolute
          SECTION   1
R1        CLR.L     D2              "R1" is simple relocatable
          ADD       D1,D3
R2        MOVE      D3,(A0)         "R2" is simple relocatable
          SECTION   2
R3        EQU       *               "R3" is simple relocatable

          MOVE      ARRAY+10,D7     absolute source operand
          MOVE      R1+10,D7        simple relocatable source operand
          MOVE      R2-R1,D7        absolute source operand
          MOVE      R1+R2,D7        complex relocatable source operand
          MOVE      R3-R2           complex relocatable source operand
```

2.8.5 Operator Precedence

Operators recognized by the assembler include the following:

a. Arithmetic operators:

addition	(+)	
subtraction	(-)	
multiplication	(*)	
division	(/)	-- produces a truncated integer result
unary minus	(-)	

b. Shift operators (binary):

shift right	(>>)	-- the left operand is shifted to the right (and zero-filled) by the number of bits specified by the right operand
shift left	(<<)	-- analogous to >>

c. Logical operators (binary):

and	(&)
or	(!)

Expressions are evaluated with the following operator precedence:

1. parenthetical expression (innermost first)
2. unary minus
3. shift
4. and, or
5. multiplication, division
6. addition, subtraction

Operators of the same precedence are evaluated left to right. All results (including intermediate) of expression evaluation are 32-bit, truncated integers. Valid operands include numeric constants, ASCII literals, absolute symbols, and relocatable symbols (with "+" and "-" only).

2.9 REGISTERS

The MC68000 has sixteen 32-bit registers (D0-D7, A0-A7) in addition to a 24-bit program counter and 16-bit status register.

Registers D0-D7 are used as data registers for byte, word, and longword operations. Registers A0-A7 are used as software stack pointers and base address registers; they may also be used for word and longword data operations. All 16 registers may be used as index registers.

Register A7 is used as the system stack pointer. (The MPU actually provides two hardware stack pointers, depending upon whether the instruction is executing in the supervisor or user state. Stack pointers and the supervisor/user states are explained under "Stacks and Queues" and "Privilege States" in the M68000 16/32-Bit Microprocessor Programmer's Reference Manual.)

(MC68010 only) The MC68010 has an additional 32-bit register (VBR) and two 3-bit registers (SFCR and DFCR). The contents of the VBR are added to the previously-calculated vector offset, during exception processing, to produce the actual vector location. The 3-bit registers allow supervisor access to other address spaces via MOVES, supplying function codes in the SFCR for the read cycle(s) from the effective address location, or supplying function codes in the DFCR for the write cycle(s) to the effective address location, respectively.

(MC68020 only) The MC68020 has four additional 32-bit registers. Two of these registers are used as stack pointers (MSP and ISP). The other two are used as cache registers (CACR and CAAR). MSP (master stack pointer) is active whenever both the "S" and "M" bits of the status register are set (supervisor state). ISP (interrupt stack pointer) is active whenever the "S" bit is set but not the "M" bit (interrupt state). USP (user stack pointer) is active whenever the "S" bit is 0 (user state). The cache registers support the onboard instruction cache of the MC68020 and can be accessed only in the supervisor state.

(MC68020 only) The assembler also provides 17 pseudo register names for the MC68020. These are the suppressed address registers used in various MC68020 addressing modes. Each register represents a content value of zero:

ZA0-ZA7	Suppressed address registers
ZD0-ZD7	Suppressed data registers
ZPC	Suppressed program counter

(MC68881 only) The MC68881 floating-point co-processor provides eight 80-bit registers and three 32-bit registers. The 80-bit registers are the floating point data registers, FP0-FP7, that serve as destinations for most floating point operations. The 32-bit registers are the system registers STATUS, CONTROL, and IADDR.

The following register mnemonics are recognized by the assembler:

D0-D7	Data registers.
ZD0-ZD7	Suppressed data registers (refer to paragraph 2.6.3 (MC68020 only))
A0-A7	Address registers.
ZA0-ZA7	Suppressed address registers (refer to paragraph 2.6.3 (MC68020 only))
A7, SP	Either mnemonic represents the system stack pointer of the active system state.

USP	User stack pointer (for user state on the MC68020).
MSP	Master stack pointer (for supervisor state on MC68020 only).
ISP	Interrupt stack pointer (for interrupt state on MC68020 only).
CCR	Condition code register (low 8 bits of SR).
SR	Status register. All 16 bits may be modified in the supervisor state. Only low 8 bits (CCR) may be modified in user state.
PC	Program counter. Used only in forcing program counter-relative addressing (refer to paragraphs 2.5.3.3 and 2.5.3.4).
ZPC	Suppressed program counter (refer to paragraph 2.6.3 (MC68020 only)).
VBR	Vector base register (MC68010 or newer only). Supports multiple vector table areas during exception processing. Accessed by the MOVEC instruction.
SFC or SFCR	Alternate function code source register (MC68010 or newer only). Accessed by the MOVEC instruction.
DFC or DFCR	Alternate function code destination register (MC68010 or newer only). Accessed by the MOVEC instruction.
CACR	Cache control register. This provides supervisor state control and status access to the onboard instruction cache (MC68020 only).
CAAR	Cache address register. This holds the address of the cache control functions requiring an address (MC68020 only).
FP0-FP7	Floating-point data registers (MC68881 only).
CONTROL	Floating-point control register. Contains four bytes. The third is the exception enable byte to enable/disable traps for each class of floating-point exception. The fourth byte is a mode byte to set the user selectable modes. The remaining bytes are reserved (MC68881 only).
STATUS	Floating-point status register contains four bytes. The first byte is a floating-point condition code byte, containing five condition codes that are set by all move and arithmetic floating-point instructions except FMOVEM. The third byte is a floating-point exception byte. The fourth byte is a floating-point accrued exception byte containing the logical inclusive OR of all floating-point exceptions that have occurred since this byte was last cleared by the user. The remaining bytes are reserved (MC68881 only).
IADDR	Floating-point instruction address register. Contains the logical address in the main processor memory of the offending instruction that generated a floating-point exception trap (MC68881 only).

2.10 INSTRUCTION MNEMONICS

The instruction operations described in paragraphs 2.10.1 through 2.10.12 are used by the assembler for MC68000, MC68010, and MC68020. Paragraphs 2.10.13 and 2.10.14 however, describe instructions which are valid only for the MC68010 and MC68020 microprocessors. Paragraphs 2.10.15 through 2.10.24 describe instructions which are valid only for the MC68020 microprocessor. Paragraphs 2.10.25 through 2.10.25.12 describe instructions which are valid only for the MC68881 floating-point co-processor.

NOTE

The M68000 Family Resident Structured Assembler has been implemented using the instructions described in this section. Differences found in the MC68020 32-Bit Virtual Memory Microprocessor Reference Manual or the MC68020 32-Bit Virtual Memory Microprocessor User's Manual will be resolved in the next revision.

2.10.1 Arithmetic Operations

The MC68000/MC68010/MC68020 instruction set includes the operations of add, subtract, multiply, and divide. Add and subtract are available for all data operand sizes. Multiply and divide may be signed or unsigned. Operations on decimal data (BCD) include add, subtract, and negate. The general form is:

```
[label:]    <operation>.<size>    <source>,<destination>
```

Examples:

ADD.W	D1,D2	Adds low order word of D1 to low order word of D2.
SUB.B	#5,(A1)	Subtracts value 5 from byte whose address is contained in A1.

On the MC68020, the signed and unsigned multiply instructions can support a 32-bit multiplier and a 64-bit product using an alternate operand syntax. This is achieved by using two data registers. One data register, Dj, holds the multiplier before multiplication and the low-order longword of the product after multiplication. Another data register, Di, holds the high-order longword of the product after multiplication (Di must be different from Dj). The general formats are:

```
MULS.L    <ea>,[Di:]Dj    (signed multiply)
MULU.L    <ea>,[Di:]Dj    (unsigned multiply)
```

where : is a required delimiter.

The signed and unsigned divide instructions on the MC68020 have been similarly expanded to support a 64-bit dividend, a 32-bit quotient, and a 32-bit remainder. This is achieved by using two data registers to hold the dividend before division and to separately hold the quotient and remainder after division. Data register Di thus holds the high-order longword of the dividend before division and the remainder after division. Data register Dj holds the low-order longword of the dividend before division and the quotient after division. If a single data register Dj is specified or Di equals Dj, it is to be used as both the high-order and low-order 32 bits of the dividend for the divide, and the 32-bit quotient of the division is returned in it. No remainder is returned in this case. The general formats are:

```
DIVS.L    <ea>,[Di:]Dj    (signed division)
DIVU.L    <ea>,[Di:]Dj    (unsigned division)
```

2.10.2 MOVE Instruction

The MOVE instruction is used to move data between registers and/or memory. The general form is:

```
MOVE.<size>    <source>,<destination>
```

where:

<size> = B, W, or L

Examples:

MOVE	D1,D2	Moves low order word of D1 into low order word of D2.
MOVE.L	XYZ,DEF	Moves longword addressed by XYZ into longword addressed by DEF.
MOVE.W	#'A',ABC	Moves word with value of $4100 into word addressed by ABC.
MOVE	ADDR,A3	Moves word addressed by ADDR into low order word of A3.

2.10.3 Compare and Check Instructions

The general formats of the compare and check instructions are:

CMP.<size> <operand$_1$>,<operand$_2$>

CHK <bounds>,<register>

where operand$_1$ is compared to operand$_2$ by the subtraction of operand$_1$ from operand$_2$ without altering operand$_1$ or operand$_2$.

The MC68020 allows the check instruction to have a longword size:

CHK.<size> <ea>,Dn

where:

<size> = W (default) or L

On the MC68020, the CMPI instruction allows any data addressing mode other than immediate for the specified effective address location.

Condition codes resulting from the execution of the compare instruction are set so that a "less than" condition means that operand$_2$ is less than operand$_1$, and "greater than" means that operand$_2$ is greater than operand$_1$.

The CHK instruction will cause a system trap if the register contents are less than zero or greater than the value specified by "bounds".

Examples:

CMP.L	ADDR,D1	Compares longword at location ADDR with contents of D1, setting condition codes accordingly.
CHK	(A0),D3	Compares word whose address is in A0 with low order word of D3; if check fails (see text), a system trap is initiated.

2.10.4 Logical Operations

Logical operations include AND, OR, EXCLUSIVE OR, NOT, and two logical test operations. These functions may be done between registers, between registers and memory, or with immediate source operands. The general form is:

<operation>.<size> <source>,<destination>

where:

<size> = B, W, or L

Example:

AND	D1,D2	Low order word of D2 receives logical 'and' of low order words in D1 and D2.

The destination may also be the status register (SR) or the condition code register (CCR) in the case of the ANDI instruction.

2.10.5 Shift Operations

Shift operations include arithmetic and logical shifts, as well as rotate and rotate with extend. All shift operations may be either fixed with the shift count in an immediate field or variable with the count in a register. Shifts in memory of a single-bit position left or right may also be done. The general form is:

<operation>.<size> <count>,<operand>

where:

<size> = B, W, or L

Examples:

LSL.W	#5,D3	Performs a left, logical shift of low order word of D3 by 5 bits; .W is optional (default).
ASR	(A2)	Performs a right, arithmetic shift of word whose address is contained in A2; because this is a memory operand, the shift is only 1 bit.
ROXL.B	D3,D2	Performs a left rotation with extend bit of low order byte of D2; shift count is contained in D3.

2.10.6 Bit Operations

Bit operations allow test and modify combinations for single bits in either an 8-bit operand for memory destinations or a 32-bit operand for data register destinations. The bit number may be fixed or variable. The general form is:

<operation> <bitno>,<operand>

where:

<size> = B or L

Examples:

BCLR	#3,XYZ(A3)	Clears bit number 3 in byte whose address is given by address in A3 plus displacement of XYZ.
BCHG	D1,D2	Tests a bit in D2, reflects its value in condition code Z, and then changes value of that bit; bit number is specified in D1.

Under Mask3 of the MC68000 chip, the instructions BCLR, BSET, and BTST have 8-bit memory operands; under Mask2 they had 16-bit memory operands. To enable users who wrote programs under Mask2 -- using BCHG, BCLR, BSET, and BTST instructions -- and to reassemble these programs under Mask3, the replacement instructions BCHGW, BCLRW, BSETW, and BTSTW are provided. These instructions align the destination operand at the next higher byte when bits 0-7 are accessed (thus functioning under Mask3 exactly as BCHG, BCLR, BSET, and BTST functioned under Mask2). In making the change, replace only the instruction mnemonic; no change is required to the operand field.

2.10.7 Conditional Operations

Condition codes can be used to set and clear data bytes. The general form is:

```
Scc     <location>
```

where "cc" may be one of the following condition codes:

CC or HS	GE	LS	PL
CS or LO	GT	LT	T
EQ	HI	MI	VC
F	LE	NE	VS

Example:

```
SNE   (A5)+
```

If condition code NE (not equal) is true, then set byte whose address is in A5 to 1's; otherwise, set that byte to 0's; increment A5 by 1.

2.10.8 Branch Operations

Branch operations include an unconditional branch, a branch to subroutine, and 14 conditional branch instructions. The general form is:

```
<operation>.<extent>     <location>
```

Examples:

```
BRA      TAG        Unconditional branch to the address TAG.
BSR      SUBDO      Branch to subroutine SUBDO.
```

Bcc.S NEXT — Short branch to NEXT on condition "cc", which may be one of the following condition codes (note that T and F are not valid condition codes for conditional branch):

CC or HS	GT	LT	VC
CS or LO	HI	MI	VS
EQ	LE	NE	
GE	LS	PL	

All conditional branch instructions use PC-relative addressing only and may be either one-word or two-word instructions. The corresponding displacement ranges are:

```
one-word     -128...+127 bytes          (8-bit displacement)
two-word     -32768...+32767 bytes      (16-bit displacement)
```

Forward references in branch instructions use the longer format by default (OPT BRW). The default may be changed to the shorter format by specifying OPT BRS or OPT BRB. The default extent may be overridden for a single branch operation by appending appropriate extent codes to the instruction -- for example:

```
BRA.S     LAB
```

A branch instruction with a byte displacement must not reference the statement which immediately follows it. This would result in an 8-bit displacement value of 0, which is recognized by the assembler as an error condition.

Example (illegal):

```
        BEQ.S   LAB1      LAB1 is the next memory word and, thus, generates
LAB1    MOVE    #1,D0     an error.
```

The MC68020 allows three sizes of offsets: byte (.B or .S), word (.W), and longword (.L). These provide byte, 2-byte, and 4-byte offsets, respectively. Compatibility with the old word (.L) sizes is available by using the new OPT OLD directive (refer to paragraph 3.5.2.10). The default offset size is still word (two bytes). Branch sizes can also be forced with several new force branch size directives: BRB or BRS (generates 8-bit defaults), BRW (generates 16-bit defaults), BRW (generates 16-bit defaults), and BRL (generates 32-bit defaults) (refer to paragraph 3.5.2.10). These new branching sizes allow the following:

```
Bcc. <size>    <label>
BRA. <size>    <label>
BSR. <size>    <label>
```

where:

<size> = B (or S), W, or L

2.10.9 Jump Operations

Jump operations include a jump to subroutine and an unconditional jump. The general form is:

```
<operation>.<extent>     <ea>
```

Examples:

JMP	4(A7)	Unconditional jump to the location 4 bytes beyond the address in A7.
JMP.L	NEXT	Long (absolute) jump to the address NEXT.
JSR	SUBDO	Jump to subroutine SUBDO.

Forward references to a label will use the long absolute address format by default (OPT FRL). The default may be changed to the shorter format by specifying OPT FRS. The default extent may be overridden on a single jump operation to a label by appending S or L as an extent code for the instruction.

2.10.10 DBcc Instruction

This instruction is a looping primitive of three parameters: condition, data register, and label. The instruction first tests the condition to determine if the termination condition for the loop has been met and, if so, no operation is performed. If the termination condition is not true, the data register is decremented by one. If the result is -1, execution continues with the next instruction. If the result is not equal to -1, execution continues at the location indicated by label. Label must be within a 16-bit displacement. The general format of the instruction is:

```
DBcc     <data register>,<label>
```

where:

"cc" may be one of the following condition codes:

CC or HS	GE	LS	PL
CS or LO	GT	LT	T
EQ	HI	MI	VC
F	LE	NE	VS

Examples:

```
LAB1   NOP
       DBGT   D0,LAB1
       DBLE   D1,LAB2
       DBT    D2,LAB1
       DBF    D3,LAB2
LAB2   NOP
```

2.10.11 Load/Store Multiple Registers

This instruction allows the loading and storing of multiple registers. Its general format is:

MOVEM.<size> <registers>,<location> (register to memory)

MOVEM.<size> <location>,<registers> (memory to register)

where:

<size> may be either W (default) or L.

The <registers> operand may assume any combination of the following:

R1/R2/R3, etc., means R1 and R2 and R3

R1-R3, etc., means R1 through R3

When specifying a register range, A and D registers cannot be mixed; e.g., A0-A5 is legal, but A0-D0 is not.

The order in which the registers are processed is independent of the order in which they are specified in the source line; rather, the order of register processing is fixed by the instruction format. For further details, refer to the MOVEM instruction in the MC68000 16/32-Bit Microprocessor Programmer's Reference Manual.

Examples:

MOVEM (A6)+,D1/D5/D7	Load registers D1, D5, and D7 from three consecutive (sign-extended) words in memory, the first of which is given by the address in A6; A6 is incremented by 2 after each transfer.
MOVEM.L A2-A6,-(A7)	Store registers A2 through A6 in five consecutive longwords in memory; A7 is decremented by 4 (because of .L); A6 is stored at the address in A7; A7 is decremented by 4; A5 is stored at the address in A7, etc.
MOVEM (A7)+,A1-A3/D1-D3	Loads registers D1, D2, D3, A1, A2, A3 in order from the six consecutive (sign-extended) words in memory, starting with the address in A7 and incrementing A7 by 2 at each step.
MOVEM.L A1/A2/A3,REGSAVE	Store registers A1, A2, A3 in three consecutive longwords starting with the location labeled REGSAVE.

2.10.12 Load Effective Address

This instruction allows computation and loading of the effective address into an address register. The general format is:

```
LEA   <operand>,<register>
```

Example:

```
LEA   XYZ(A2,D5),A1        Load A1 with effective address specified by
                           the first operand.  Refer to paragraph
                           2.5.2.5 for an explanation of addressing mode
                           "Address Register Indirect With Index".
```

2.10.13 Move to/from Control Register

(MC68010 or newer only) With this instruction, the specified control register is copied to the specified general register, or the specified general register is copied to the specified control register. This is always a 32-bit transfer, even though the control register may be implemented with fewer bits. Unimplemented bits read as zeros. The general format is:

```
MOVEC   <control register>,<register>
MOVEC   <register>,<control register>
```

Examples:

```
MOVEC     VBR,A0           Copies contents of vector base register to
                           register A0.

MOVEC     D7,SFC           Copies contents of register D7 to the source
                           function code register (3 bits).

MOVEC     DFC,D0           Copies contents of destination function code
                           register to register D0 (3 bits; zero filled).

MOVEC.L   USP,D7           Copies user stack pointer to register D7.
```

(MC68020 only) Four additional <control register> values are recognized for the MC68020:

```
CACR:    Cache control register
CAAR:    Cache address register
MSP:     Master stack pointer
ISP:     Interrupt stack pointer
```

NOTE

If the instruction is used without setting the processor type in this command line or in the OPT P=68XXX, then an error is generated.

2.10.14 Move to/from Address Space

(MC68010 or newer only.) Moves a byte, word, or longword from the specified general register to a location within the address space determined by the DFC register, or moves a byte, word, or longword from a location within the address space determined by the SFC register to the specified general register. Note that with a byte operation size specified, the address register direct mode is not allowed.

General format:

```
MOVES <ea>,<register>
MOVES <register>,<ea>
```

Examples:

MOVES.W	(A2)+,D2	Moves a word at the address contained in register A2 to register D2 and then increments A2 by 2.
MOVES	A4,LABEL	Moves the lower word of register A4 to the address of LABEL.
MOVES	2222,A2	Moves one word of data beginning at address 2222 to register A2.

NOTE

If the instruction is used without setting the processor type in the command line or in the OPT P=68XXX, then an error is generated.

2.10.15 Bit Fields and Instructions (MC68020 only)

NOTE

The instructions in paragraphs 2.10.15 through 2.10.24 require that the processor type be set to 68020 in the command line or in the OPT P=68XXX directive. Otherwise, an error is generated.

A bit field is a string of consecutive bits in a bit array. The address of the bit array is determined by the address of the byte containing bit 0 (the base address). Bit fields extend in both directions from bit 0 and are assigned bit field numbers from 0 (the leftmost and most significant bit) to 7 (the rightmost and least significant bit). By this notation a preceding byte's least significant bit has a bit field number of 8. Instructions reference bit fields using two parameters: a bit field offset and a bit field width. A bit field offset is the bit field number of the leftmost bit in the field; its range is -2**31 to (2**31) - 1. The bit field width is the number of bits in the bit field; its range is 1 to 32.

2.10.15.1 Single Operand Bit Field Instructions.

This section explains the following single operand bit fields: complement, clear, set and test.

Complement Bit Field

Complements a bit field at the specified effective address location. Condition code fields are modified, depending on the value in the bit field before the complement. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

General format:

```
BFCHG  <ea>{<offset>:<width>}
```

where:

{, :, and } are required delimiters; <offset> and <width> may be an immediate value or a data register D0-D7.

Example:

```
BFCHG  LABEL{0:D1}  Complements the bit field at address LABEL from bit
                    0 to bit n - 1 where n is the value in D1.
```

Clear Bit Field

Clears a bit field at a specified effective address location. Condition code fields are modified, depending on the value in the bit field before the clear. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

General format:

```
BFCLR  <ea>{<offset>:<width>}
```

where:

{, :, and } are required delimiters; <offset> and <width> may be an immediate value or a data register D0-D7.

Example:

```
BFCLR  LABEL{D1:8}  Clears the bit field at address LABEL starting at the
                    bit specified in D1 for 8 bits.
```

Set Bit Field

Sets all bits of a bit field at a specified effective address location. Condition code fields are modified, depending on the value in the bit field before the set. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

General format:

```
BFSET  <ea>{<offset>:<width>}
```

where:

{, :, and } are required delimiters; <offset> and <width> may be an immediate value or a data register D0-D7.

Example:

```
BFSET  2222{0:8}     Sets one byte at address 2222 to all 1's.
```

Test Bit Field

Sets condition codes according to the value in the bit field at the specified effective address location. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).
General format:

```
BFTST  <ea>{<offset>:<width>}
```

where:

{, :, and } are required delimiters; <offset> and <width> may be an immediate value or a data register D0-D7.

Example:

```
BFTST  (A2){D1:D2}  Clears condition codes V and C; sets N and Z
                    according to the bits of the bit field.
```

2.10.15.2 Double Operand Bit Field Instructions.

This section explains the following double operand bit fields: extract signed, extract unsigned, find first one, and insert.

Extract Bit Field Signed

The bit field at the specified effective address location is sign-extended to 32 bits and loaded into a data register. Condition code fields are modified, depending on the value in the bit field before the sign extension. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

General format:

```
BFEXTS  <ea>{<offset>:<width>},Dn
```

where:

{, :, and } are required delimiters; <offset> and <width> may be an immediate value or a data register D0-D7.

Example:

```
BFEXTS  LABEL{0:8},D1      The value of the byte at LABEL is sign-extended
                           and then loaded into D1.
```

Extract Bit Field Unsigned

The bit field at the specified effective address location is zero extended to 32 bits and loaded into a data register. Condition code fields are modified, depending on the value in the bit field before the zero extension. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

General format:

```
BFEXTU  <ea>{<offset>:<width>},Dn
```

where:

{, :, and } are required delimiters; <offset> and <width> may be an immediate value or a data register D0-D7.

Example:

```
BFEXTU  LABEL{0:8},D1      The value of the byte at LABEL is zero-extended
                           and then loaded into D1.
```

Find First One in Bit Field

The bit field at the specified effective address location is examined for the most significant bit position that is set. If a set bit exists, the bit offset for that bit is loaded into a data register. If no bit is set, a value is loaded in a data register equal to the offset plus the width of the bit field. Condition code fields are modified, depending on the value in the bit field before the examination. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

General format:

```
BFFFO  <ea>{<offset>:<width>},Dn
```

where:

{, :, and } are required delimiters; <offset> and <width> may be an immediate value or a data register D0-D7.

Example:

```
BFFFO  LABEL{0:1},D1      If bit 0 is set, then 0 is loaded into D1, else 1
                          is loaded into D1.
```

Insert Bit Field

Move a bit field from the low-order bits of the data register to the bit field at the specified effective address location. Condition code fields are modified, depending on the value in the bit field before the insertion. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

General format:

```
BFINS  Dn,<ea>{<offset>:<width>}
```

where:

{, :, and } are required delimiters; <offset> and <width> may be an immediate value or a data register D0-D7.

Example:

```
BFINS  D1,LABEL{0:8}      Move the low-order byte of D1 to the 8-bit field at
                          address LABEL.
```

2.10.16 Check Instructions (MC68020 only)

Check and compare register instructions are discussed in the following paragraphs.

2.10.16.1 Check Register Against Bounds. Check the value in register Rn against the lower- and upper-bound pair at the address of the specified effective address location. When signed comparisons are made, the arithmetically smaller value should be taken as the lower bound. For unsigned comparisons, the logically smaller value should be taken as the lower bound. If the instruction size is byte (.B) or word (.W), only the low-order byte or word of a data register, respectively, is used for the comparison. When Rn is an address register, an instruction size of byte or word results in a sign extension of the bound operands before the comparison. If the register is out of bounds, exception processing is initiated and a vector number that references the CHK instruction exception vector is generated. Only control addressing modes are allowed.

General format:

```
CHK2.<size>  <ea>,Rn
```

where:

<size> = B, W, or L

Example:

```
CHK2.B  LABEL,D1
```

The instruction following this instruction is executed provided the value of the low-order byte of D1 is greater than the contents of LABEL and less than the contents of LABEL + 1.

2.10.16.2 Compare Register Against Bounds. This has exactly the same functionality as CHK2, except that no exception processing will occur if the value in Rn is not within bounds.

General format:

```
CMP2  <ea>,Rn
```

2.10.17 Truncated Divide Instructions (MC68020 only)

Truncated divide instructions use signed or unsigned arithmetic.

2.10.17.1 Truncated Signed Divide. Using signed arithmetic, divide the 32-bit value in data register Dj by the value at the specified effective address location. After division, Dj contains the signed 32-bit quotient, and Di optionally contains the 32-bit remainder, provided Di is not equal to Dj. When Di and Dj are the same, no remainder is generated.

General format:

```
TDIVS.<size>  <ea>,[Di:]Dj
```

where:

```
<size> = L
```

2.10.17.2 Truncated Unsigned Divide. Using unsigned arithmetic, divide the 32-bit value in data register Dj by the value at the specified effective address location. After division, Dj contains the unsigned 32-bit quotient and data register, and Di optionally contains the 32-bit remainder, provided Di is not equal to Dj. When Di and Dj are the same, no remainder is generated.

General format:

```
TDIVU.<size>  <ea>,[Di:]Dj
```

where:

```
<size> = L
```

2.10.18 Sign Extend Instructions (MC68020 only)

Instructions are given for both byte and word.

2.10.18.1 Sign Extend Byte. Extend bit 7 to bits 31 through 8 of data register Dn if <size> is long, or else extend bit 7 to bits 15 through 8.

General format:

EXTB.<size> Dn

where:

<size> = W (default) or L

Example:

EXTB.L D1 Copies the value of bit 7 to bits 31 through 8 of D1.

NOTE

EXTB.W is the same as the instruction
EXT.W for the MC68000 and MC68010.

2.10.18.2 Sign Extend Word. Extend bit 15 to bits 31 through 16 of data register Dn.

General format:

EXTW.<size> Dn

where:

<size> = W (default) or L

NOTE

EXTB.W is the same as the instruction
EXT.W for the MC68000 and MC68010.

2.10.19 BCD Instructions (MC68020 only)

Instructions include both pack and unpack for BCD.

2.5.19.1 Pack BCD. Pack the low four bits of each of two bytes into one byte. When a predecrement addressing mode is specified, bits 3 through 0 of the two fetched consecutive source bytes are concatenated to form a packed byte, which is written to the destination. When both operands are data registers, bits 11 through 8 and bits 3 through 0 of the source register are concatenated to form the low-order packed byte of the destination data register.

General format:

```
PACK   -(Ay),-(Ax)
PACK   Dy,Dx
```

where:

Ay and Ax are address registers; Dy and Dx are data registers

2.10.19.2 Unpack BCD. Two BCD digits in the source byte are stored in two consecutive bytes at the destination. When predecrement addressing is used, two BCD digits in the source byte are separately written to two consecutive bytes, and destination bits 7 through 4 are set to zero. When data registers are used, bits 7 through 4 and bits 3 through 0 of the source register are placed in bits 11 through 8 and bits 3 through 0, respectively, of the destination register. All other bits of the destination register are set to zero.

General format:

```
UNPK   -(Ay),-(Ax)
UNPK   Dy,Dx
```

where:

Ay and Ax are address registers; Dy and Dx are data registers.

2.10.20 Module Instructions (MC68020 only)

Both call and return are module instructions.

2.5.20.1 Call Module. An external module descriptor resides at the specified effective address location. This module descriptor contains control information for entry into the associated module. A module frame, containing the current module state, is created on the top of the stack. The new module state is then loaded from the external module descriptor. No condition codes are affected by this instruction. Only control-alterable addressing is allowed.

General format:

```
CALLM   #ddd,<ea>
```

where:

ddd is the 8-bit number of bytes of arguments passed to the called module.

2.10.20.2 Return from Module. A previously saved module state, from a CALLM instruction, is reloaded from the top of the stack. A register Rn is used as the module data area pointer. If the module state includes a saved module data area pointer, register Rn is restored; else Rn is unchanged. No condition codes are affected.

General format:

```
RTM   Rn
```

2.10.21 Trap on Condition Code (MC68020 only)

If the specified condition code is true, exception processing occurs. The vector number generated references the TRAPcc exception vector. The stacked program counter points to the next instruction. If the specified condition code is false, control passes to the next instruction in sequence. Any of the 16 condition codes may be referenced. Condition codes are not affected by this instruction. The #xxx parameter in the TPcc form allows a 16- or 32-bit value to be embedded within the instruction for reference in exception processing.

General format:

```
Tcc
TPcc.<size>   #xxx
```

where:

<size> = W (default) or L

2.10.22 Compare and Swap with Operand (MC68020 only)

In the CAS instruction the specified effective address location is compared to data register Dw. If operands match, the update operand in data register Dy is written to the specified effective address location. If operands do not match, Dw is loaded with the contents at the specified effective address location.

In the CAS2 instruction, both Dw1 and Dw2 must match the values in memory pointed to by registers Rz1 and Rz2, respectively, in order that these memory values be updated by the contents of registers Dy1 and Dy2. If both operands do not match, then Dw1 and Dw2 are loaded with the contents of memory pointed to by registers Rz1 and Rz2, respectively.

General formats:

```
CAS.<size>   Dw,Dy,<ea>

CAS2.<size>  Dw1:Dw2,Dy1:Dy2,(Rz1):(Rz2)
```

where:

<size> = B, W, or L

2.10.23 Breakpoint (MC68020 only)

The operation of this instruction is implementation-dependent. The processor will ask for the operation word which the breakpoint has replaced. If the operation word is furnished, the processor will execute that instruction and continue. If the operation word is not furnished, the processor will take an illegal instruction exception.

General format:

```
BKPT  #<vector>
```

where:

#<vector> specifies the breakpoint for which the processor is to request the corresponding operation word. Value = 0 through 7.

2.10.24 The MC68881 Co-Processor Instructions (MC68881 only)

NOTE

The instructions in paragraphs 2.10.25.1 and 2.10.25.2 require that the processor type be set to 68881 in the OPT directive or from the command line. Otherwise, an error is generated.

At present, the assembler supports only the MC68881 co-processor. The assembler syntax that follows refers to opcodes for the MC68881, even though the MC68020 can support other co-processors which follow MC68881 protocols. Floating-point condition codes used in opcodes for the MC68881 are listed in Table 2-11.

TABLE 2-11. MC68881-Specific Floating-Point Condition Codes (fpcc)

TRAP ON UNORDERED

fpcc		INVERSE	
GT	Greater than	NGT	Not greater than
GE	Greater than or equal	NGE	Not greater than or equal
LT	Less than	NLT	Not less than
GL	Greater or less than	NGL	Not greater or less than
LE	Less than or equal	NLE	Not less than or equal
GLE	Greater or less than or equal	NGLE	Not greater or less than or equal
SEQ	Equal	SNEQ	Not equal
ST	Always	SF	Never

NO TRAP ON UNORDERED

fpcc		INVERSE	
OGT	Greater than	ULE	Not greater than (Unordered or less or equal)
OGE	Greater than or equal	ULT	Not greater than or equal (Unordered or less than)
OLT	Less than	UGE	Not less than (Unordered or greater or equal)
OGL	Greater or less than	UEQ	Not greater or less than (Unordered or equal)
OLE	Less than or equal	UGT	Not less than or equal (Unordered or greater than)
OR	Ordered	UN	Unordered
EQ	Equal	NEQ	Not equal (Unordered or greater or less)
T	Always	F	Never

2.10.24.1 Co-Processor Branch Conditionally. If the specified floating-point condition code is met, program execution continues at address <label>. Condition codes are not affected by this instruction.

General format:

FBfpcc <label>

where:

fpcc is defined in paragraph 2.10.24.

2.10.24.2 Decrement and Branch on Condition. If the specified floating-point condition code is met, execution continues with the next instruction. Otherwise, the low-order word in the specified data register Dn is decremented by one. If the result is equal to -1, execution continues with the next instruction, else execution continues at address <label>. Condition codes are not affected by this instruction.

General format:

FDBfpcc Dn,<label>

where:

fpcc is defined in paragraph 2.10.25.

2.10.24.3 Set on Condition. The specified floating-point condition code is tested. If the condition is true, the byte at the specified address location is set to TRUE (all 1's); otherwise that byte is set to FALSE (all 0's). No condition codes are affected by this instruction. Only data-alterable addressing modes are allowed.

General format:

FSfpcc <ea>

where:

fpcc is defined in paragraph 2.10.25.

2.10.24.4 Trap on Condition, with or without a parameter. If the selected floating-point condition code is true, then the processor initiates exception processing. The vector number generated references the TRAPcc exception vector. The stacked Program Counter then points to the next instruction. If the selected floating-point condition code is not true, then no operation is performed and execution continues with the next instruction in sequence. Condition codes are not affected by this instruction.

General formats:

```
FTfpcc
FTPfpcc.<size>   #xxx
```

where:

fpcc is defined in paragraph 2.10.25.
<size> = W or L
#xxx is a 16- or 32-bit parameter used to uniquely identify a particular FTPfpcc instruction.

2.10.24.5 Co-processor Save Function. This instruction saves the internal state for the context switch at the specified effective address location. Condition codes are not affected by this instruction. Only postdecrement or alterable control addressing modes are allowed. This is a privileged instruction.

General format:

```
FSAVE   <ea>
```

2.10.24.6 Restore Internal State of Co-Processor. This instruction restores the internal state for the context switch from the specified effective address location. Condition codes are not affected by this instruction. Only postincrement or control addressing modes are allowed. This is a privileged instruction.

General format:

```
FRESTORE   <ea>
```

2.10.24.7 Move to Floating-Point Register from Memory or from Another Floating-Point Register Instruction.

```
FMOVE.<size>    <ea>,FPn
```

where:

<size> = B, W (default), L, S, D, X, or P; FPn is a floating-point register

```
FMOVE.<size>     FPm,FPn
FMOVEM.<size>    <ea>,<fp_reg_list>
FMOVEM.<size>    <ea>,Dn                (See note below.)
FMOVECR.<size>    #ccc,FPn
```

where:

<size> = X (default); FPm, FPn are different floating-point registers; <fp_reg_list> is of the form FP1/FP2/FP3... and/or FP1-FP3...; Dn is a data register; ccc is a group of frequently used floating-point constants: (These are tentative.)

ccc	value
00	Pi = 3.14159...
0B	Log10(2)
0C	e = 2.71828...
0D	Log2(e)
0E	Log10(e)
0F	0.0
10	Logn(2)
11	Logn(10)
12	10**0 (1.0)
13	10**1
14	10**2
15	10**4
16	10**8
17	10**16
18	10**32
19	10**64
1A	10**128
1B	10**256
1C	10**512
1D	10**1024
1E	10**2048
1F	10**4096

NOTE: Dn indicates that the FMOVEM bit mask is in a MC68020 data register.

```
FMOVEM.<size>          <ea>,CONTROL/STATUS/IADDR     (See note below.)
FMOVE.<size>           <ea>,CONTROL|STATUS|IADDR
```

where:

<size> = L (default); CONTROL, STATUS, and IADDR are floating-point system registers.

NOTE: From 1 to 3 of these registers can be specified, each separated from one another by a slash in FMOVEM. The vertical lines of FMOVE represent selection choices.

2.10.24.8 Move from Floating-Point Register to Memory Instructions.

FMOVE.<size> FPn,<ea>

where:

<size> = B, W (default), L, S, D, X, or P; FPn is a floating-point register.

FMOVE.P FPn,<ea>{#k}

where:

FPn is a floating-point register; { and } are required delimiters; #k has a default value of immediate data -16.

FMOVE.P FPn,<ea>{Dn}

where:

FPn is a floating-point register; { and } are required delimiters; Dn is a data register holding a dynamic k value.

FMOVEM.<size> <fp_reg_list>,<ea>
FMOVEM.<size> Dn,<ea> (See note below.)

where:

<size> = X (default); <fp_reg_list> is of the form FP1/FP2/FP3... and/or FP1-FP3...; Dn is a data register.

NOTE: Dn indicates that the FMOVEM bit mask is in a MC68020 data register.

FMOVEM.<size> CONTROL/STATUS/IADDR,<ea> (See note below.)
FMOVE.<size> CONTROL|STATUS|IADDR,<ea>

where:

<size> = L (default); CONTROL, STATUS, and IADDR are floating-point system registers.

NOTE: From 1 to 3 of these registers can be specified, each separated from one another by a slash. The vertical lines of FMOVE represent selection choices.

2.10.24.9 Floating-Point Functions.

a. Source Operand in Memory

F<op>.<size> <ea>,FPn

where:

<size> = B, W (default), L, S, D, X, or P; <op> is defined below in paragraph c.; FPn is a floating-point register

b. Source Operand in Floating-Point Register

F<op>.<size> FPm,FPn

where:

<size> = B, W (default), L, S, D, X, or P; <op> is defined below in paragraph c.; FPm and FPn are floating-point registers

c. Source Operand and Destination in Same Floating-Point Register

F<op>.<size> FPn

where:

<size> = X (default); FPn is a floating-point register; <op> is:

ABS	absolute value
ACOS	arccosine
ASIN	arcsine
ATAN	arctangent
ATANH	hyperbolic arctangent
COS	cosine
COSH	hyperbolic cosine
ETOX	e**x ; powers of e (Euler's constant)
ETOXM1	e**(x-1) ; Euler's constant to the x-1 power
GETMAN	get the mantissa
GETEXP	get the exponent
INT	integer part
LOGN	natural log; base e log
LOGNP1	natural log (x+1)
LOG10	common log; base 10 log
LOG2	binary log; base 2 log
NEG	negate
SIN	sine
SINH	hyperbolic sine
SQRT	square root
TAN	tangent
TANH	hyperbolic tangent
TENTOX	10**x ; powers of 10
TWOTOX	2**x; powers of 2

d. Sine/Cosine Function

FSINCOS.<size> <ea>,FPm:FPn (FPm=sin,FPn=cos)

where:

<size> = B, W, L, S , D, X, or P
FPm is the floating-point register holding the sine result.
FPn is the floating-point register holding the cosine result.

2.10.24.10 Floating-Point Arithmetic Operations.

a. One Source Operand is in Memory

F<op>.<size> <ea>,FPn

where:

<size> = B, W (default), L, S, D, X, or P (except D or X not allowed for <size> of SGLDIV or SGLMUL); FPn is a floating-point register; <op> is defined below in paragraph b.

b. Both Source Operands in Floating-Point Registers

F<op>.<size> FPm,FPn

where:

<size> = X (default) (except only S allowed for <size> of SGLDIV or SGLMUL); FPm and FPn are floating-point registers; <op> is:

ADD	add
CMP	compare
DIV	divide
MOD	modulo
MUL	multiply
REM	remainder
SCALE	scale exponent
SGLDIV	single-precision divide
SGLMUL	single-precision multiply
SUB	subtract
YTOX	y**x (powers of y)

2.10.24.11 Floating-Point NO-OP. This instruction is supplied for synchronization with the floating-point co-processor.

General format:

```
FNOP
```

2.10.24.12 Floating-Point Test of an Operand. The operand at the specified effective address location is compared with zero. Floating-point condition codes as specified in paragraph 2.10.25 are set according to the result of the test.

General format:

```
FTEST.<size>   <ea>
```

where:

<size> = B, W, L, S, D, X, or P

2.11 VARIANTS ON INSTRUCTION TYPES

Certain instructions allow a "quick" and/or an "immediate" form when immediate data within a restricted size range appear as an operand. These abbreviated forms are normally chosen by the assembler, when appropriate. However, it is possible for the programmer to "force" such a form by appending a Q or I to the mnemonic opcode (to indicate "quick" or "immediate", respectively) on instructions for which such forms exist. If the specified quick or immediate form does not exist, or if the immediate data does not conform to the size requirements of the abbreviated form, an error is generated.

Some instructions also have "address" variant forms (which refer to address registers as destinations); these variants append an A to the instruction mnemonic (for example, ADDA, CMPA). This variant is chosen by the assembler without programmer specification, when appropriate to do so; the programmer need specify only the general instruction mnemonic. However, the programmer may "force" or specify such a variant form by appending the A. If the specified variant does not exist or is not appropriate with the given operands, an error is generated.

The CMP instruction also has a memory variant form (CMPM) in which both operands are a special class of memory references. The CMPM instruction requires postincrement addressing of both operands. The CMPM instruction will be selected by the assembler, or it may be specified by the programmer.

The variations -- A, Q, I, and M -- must conform to the following restrictions:

A — Must specify an address register as a destination, and cannot specify a byte size code (.B).

Q — Requires immediate operand be in a certain size range. MOVEQ also requires longword data size.

I — The size of immediate data is adjusted to match size code of operation.

M — Both operands must be postincrement addresses.

For example, the instruction

```
ADDQ   #9,D0        Attempts to add value 9 to D0
```

causes an assembly error, because the immediate operand is not in the valid size range (1 through 8).

Although the assembler selects the appropriate opcode variation -- A, Q, I, or M -- when the suffix is not specified, the explicit encoding of the suffix with the basic opcode is recommended for the following purposes:

a. For documentation, to make clear in the source language the instruction form that was assembled.

b. To force a format other than that which the assembler selects. For example, the assembler selects the quick (Q) form for the instruction

```
ADD     #1,D4          Adds the value 1 to D4 via an ADDQ (2-byte)
                       instruction.
```

If the immediate (I) form is desired, the programmer must declare it explicitly, as follows:

```
ADDI    #1,D4          Adds the value 1 to D4 via an ADDI (4-byte)
                       instruction.
```

c. To generate invariant code when using variant immediate data (separate assemblies).

CHAPTER 3

ASSEMBLER DIRECTIVES

3.1 INTRODUCTION

All assembler directives (pseudo-ops), with the exception of "DC" and "DCB", are instructions to the assembler rather than instructions to be translated into object code. This chapter contains descriptions and examples of the basic forms of the most frequently used assembler directives. Directives controlling the macro and conditional assembly capabilities are described in Chapter 5. Directives used in structured syntax are described in Chapter 6. The most commonly used directives supported by the assembler are grouped, by function, in Table 3-1.

TABLE 3-1. M68000 Family Assembler Directives

DIRECTIVE	FUNCTION
ASSEMBLY CONTROL	
END	Program end
INCLUDE	Include second file
MASK2	Assemble for Mask2 (R9M)
OFFSET	Define offsets
ORG	Absolute origin
SECTION	Relocatable program section
SYMBOL DEFINITION	
EQU*	Assign permanent value
FEQU*	Assign permanent floating-point value (MC68881 only)
REG*	Define register list
SET*	Assign temporary value
DATA DEFINITION/ STORAGE ALLOCATION	
COMLINE**	Command line
DC**	Define constants
DCB**	Define constant block
DS**	Define storage

TABLE 3-1. M68000 Family Assembler Directives (cont'd)

DIRECTIVE	FUNCTION
LISTING CONTROL AND OUTPUT OPTIONS	
FAIL	Programmer-generated error
FOPT	Assigns floating-point options (MC68881 only)
FORMAT	Enable the automatic formatting
NOFORMAT	Disable the automatic formatting
LIST	Enable the listing
NOLIST or NOL	Disable the listing
LLEN n	Set line lengths 72 $\leq$ n $\leq$ 132
NOOBJ	Disable object output
OPT	Assembler options
PAGE	Top of page
NOPAGE	Disable paging
SPC n	Skip n lines
TTL	Up to 60 characters of title
LINKAGE EDITOR CONTROL	
IDNT*	Relocatable identification record
XDEF	External symbol definition
XREF	External symbol reference

* Labels required.
** Label optional.

3.2 ASSEMBLY CONTROL

3.2.1 END - Program End

FORMAT: END [<start address>]

DESCRIPTION: END directive indicates to the assembler that the source is finished. Subsequent source statements are ignored. The END directive encountered at the end of the first pass through the source program causes the assembler to start the second pass. The start address should be specified unless it is external to the module. If no start address is specified, it is still possible to include a comment field, provided the comment field is set off by an exclamation point (!). This syntax indicates to the assembler that the operand field is null and that a comment field follows.

3.2.2 INCLUDE - Include Secondary File

FORMAT: INCLUDE <file spec>

DESCRIPTION: This directive is inserted in the source program at any point where a secondary file is to be included in the source input stream.

NOTE

<file spec> is case-sensitive in the SYSTEM V/68 environment.

3.2.3 MASK2 - Assemble for MASK2 (MC68000 only)

FORMAT: MASK2

DESCRIPTION: The MASK2 directive indicates that the source program is to be assembled to run on the Mask2 (R9M) chip. Specifying MASK2 implements the following changes in assembler processing:

(a) DCNT instruction replaces DBcc
(b) STOP does not take an operand
(c) Bit operations are adjusted to the R9M format

3.2.4 OFFSET - Define Offsets

FORMAT: OFFSET <expression>

DESCRIPTION: The OFFSET directive is used to define a table of offsets via the Define Storage (DS) directive without passing these storage definitions on to the linkage editor, in effect creating a dummy section. Symbols defined in an OFFSET table are kept internally, but no code-producing instructions or directives may appear. SET, EQU, REG, XDEF, and XREF directives are allowed.

<expression> is the value at which the offset table is to begin. The expression must be absolute and may not contain forward, undefined, or external references.

OFFSET must be terminated by an ORG or SECTION directive before further code-producing instructions are generated. If not, the assembler produces an error message.

3.2.5 ORG - Absolute Origin

FORMAT: ORG[.<qualifier>] <expression> [<comments>]

DESCRIPTION: The ORG directive changes the program counter to the value specified by the expression in its operand field. Subsequent statements are assigned absolute memory locations starting with the new program counter value. <expression> must be absolute and may not contain any forward, undefined, or external references.

Qualifier may be either "S" or "L". "ORG.S" is interpreted as both "ORG" and "OPT FRS" (Forward Reference Short Option). "ORG.L" is interpreted as both "ORG" and "OPT FRL" (Forward Reference Long Option). Regardless of the forward reference option, references to previously-defined absolute symbols will always generate the appropriate short or long addressing form, based upon the size of a symbol's absolute address.

3.2.6 SECTION - Relocatable Program Section

FORMAT: [<name>] SECTION[.S] <number>

DESCRIPTION: This directive causes the program counter to be restored to the address following the last location allocated in the indicated section (or to zero if used for the first time).

<name> indicates a named common area within the indicated section. No unnamed common section is allowed. <name> is associated with the section and may be reused in other sections.

".S" indicates the section should be placed in low address memory, so that direct addressing may be implemented through the absolute short mode. This information is passed on to the linkage editor. It affects the choice of address modes in certain situations where the assembler must choose between absolute short and absolute long.

<number> must be in the range 0..15. No section numbers are reserved in any way. (refer to the M68000 Family Linkage Editor User's Manual or the SYSTEM V/68 Linkage Editor User's Manual for a discussion of default assignment of sections to segments.) By default, the assembler begins with section 0.

3.3 SYMBOL DEFINITION

Symbol definition directives EQU, REG, SET , and FEQU provide the only method by which a symbol appearing in the label field may be assigned a 'value' other than that corresponding to the current location counter.

3.3.1 EQU - Equate Symbol Value

FORMAT: <label> EQU <expression> [<comments>]

DESCRIPTION: EQU directive assigns the value of the expression in the operand field to the symbol in the label field. The label and expression follow the rules given in Chapter 2. The label and operand fields are both required, and the label cannot be defined anywhere else in the program.

The expression in the operand field of an EQU cannot include a symbol that is undefined or not yet defined (no forward references are allowed). Also, it cannot be a complex relocatable expression.

3.3.2 FEQU - Equate Floating Point Symbol Value (MC68881 only)

FORMAT: <label> FEQU.<size> <value> [<comments>]

where <size> = S, D, X, or P

DESCRIPTION: FEQU directive assigns the floating-point value in the operand field to the symbol in the label field. The label and value follow the rules given in Chapter 2. The operand fields are both required, and the label cannot be defined anywhere else in the program. Note that <value> is stored as a string and converted only to its binary format when it is used in instructions. <value> may be a floating-point decimal string or a floating point hexadecimal value as defined in paragraph 2.8.1. A warning is generated whenever the number of bits required to represent the specified precision is exceeded. The subsequent <label> must not be used as an address.

3.3.3 REG - Define Register List

FORMAT: <label> REG <reg list> [<comment>]

DESCRIPTION: REG directive assigns a value to <label> that can be translated into the register list mask format used in the MOVEM instruction. The label cannot be redefined as a Class 2 symbol anywhere else in the program. <reg list> is of the form:

R1[-R2][/R3[-R4]]...

Example: A1-A5/D0/D2-D4/D7

3.3.4 SET - Set Symbol Value

FORMAT: <label> SET <expression> [<comments>]

DESCRIPTION: SET directive assigns the value of the expression in the operand field to the symbol in the label field. Thus, the SET directive is similar to the EQU directive. However, the SET directive allows the symbol in the label field to be redefined by other SET directives in the program. The label and operand fields are both required.

The expression in the operand field of a SET cannot include a symbol that is undefined or not yet defined (no forward references are allowed), nor can it be a complex relocatable expression.

3.4 DATA DEFINITION/STORAGE ALLOCATION

The directives in this section provide the only means by which object code may begin or end on odd byte boundaries. All instructions and all word or long word-sized data must begin and end on even byte boundaries. Odd byte alignment is allowed only for the DC.B, DS.B, DCB.B, and COMLINE directives. All other operations which generate object code are preceded by a zero fill byte if word boundary alignment is required.

3.4.1 COMLINE - Command Line

FORMAT: [<label>] COMLINE <expression>

DESCRIPTION: Identical to DS.B (define storage in bytes), except that it is passed on to the linkage editor as the location of the command line. <expression> is the number of bytes to reserve (>0). It must be absolute and may not contain forward, undefined, or external references. An example of use would be to pass a filename for the program to access.

3.4.2 DC - Define Constant

FORMAT:

[<label>]	DC.B	<operand(s)>	Define constant in bytes
[<label>]	DC.W	<operand(s)>	Define constant in words (default)
[<label>]	DC.L	<operand(s)>	Define constant in long words
[<label>]	DC.S	<operand(s)>	Define constant in single precision floating-point (MC68881 only)
[<label>]	DC.D	<operand(s)>	Define constant in double precision floating-point (MC68881 only)
[<label>]	DC.X	<operand(s)>	Define constant in extended precision floating-point (MC68881 only)
[<label>]	DC.P	<operand(s)>	Define constant in packed binary coded decimal (MC68881 only)

DESCRIPTION: The function of the DC directive is to define a constant in memory. The DC directive may have one or more operands, which are separated by commas. The operand field may contain the actual value (decimal, hexadecimal, or ASCII). Alternatively, the operand may be a symbol or expression which can be evaluated either by the assembler or the linker. The constant is aligned on a word boundary if word (.W), longword (.L), single precision (.S), double precision (.D), extended precision floating-point (.X), or packed BCD (.P) is specified. Alignment is on a byte boundary if byte (.B) is specified. Only byte (.B) constants may not be relocated by the linker.

The following rules apply to size specifications on DC directives with ASCII strings as operands:

DC.B One byte is allocated per ASCII character.

DC.W The string begins on a word boundary. If the string address contains an odd number of characters, a zero fill byte follows the last character.

DC.L The string begins on a word boundary. If the string length is not a multiple of four bytes, the last longword is zero filled.

Unless option CEX is in effect, a maximum of six bytes of constants is displayed on the assembly listing.

3.4.2.1 Examples of ASCII Strings

```
DC.B 'ABCDEFGHI'    Memory has nine contiguous bytes with the ASCII characters
                    A through I.

DC.B 'E'            Memory has characters "EJ" ($454A) in contiguous bytes.
DC.B 'J'

DC.B 'E'            Memory has $45004500 in contiguous bytes, the first zero
DC.W 'E'            byte being an odd byte fill as outlined above.

DC   'X'            Memory has $5800 in contiguous bytes.

DC.L '12345'        Memory has $3132333435000000 in contiguous bytes.
```

3.4.2.2 Examples of Numeric Constants

```
DC.B  10,5,7        Memory has three contiguous bytes with the decimal values
                    10, 5, and 7 in their respective bytes.

DC.W  10,5,7        Each operand is contained in a word. The value 10 is
                    contained in the first word, right justified. The value 5
                    is in the second word, and the value 7 is in the third
                    word.

DC.L  10,5,7        Each operand is contained in a longword. The value 10 is
                    contained in the first longword (4 bytes) right justified.
                    The value 5 is in the second longword, and the value 7 is
                    in the third longword.

DC    LABEL+1       The generated value is the address of LABEL plus 1 in a
                    word size operand.

DC    $FF,$10,$AE   Rules for hexadecimal are same as decimal.

DC.S  3.1415        A single precision floating-point value is created.

DC.D  2.54          A double precision floating-point value is created.
```

```
DC.X  6.0224E23    An extended precision floating-point value is created.

DC.X  :BABE10      An extended precision floating-point hex value is created.
                   NOTE: "E" here can be only a hex digit, not an exponent
                   designator.

DC.P  3.00E9       A packed BCD value is created.
```

If the resulting value in an operand expression exceeds the size of the operand, an error is generated. For example,

```
DC.B $FFF          This causes an error because $FFF cannot be repre-
                   sented in 8 bits.

DC   $FFF6F        This causes an error because $FFF6F cannot be represented
                   in 16 bits.
```

3.4.3 DCB - Define Constant Block

FORMAT: [<label>] DCB[.<size>] <length>,<value> [<comment>]

where:

<size> = B, W, L, S, D, X, or P (S, D, X, P for MC68020/MC68881 only)

<value> = <binary,decimal> (Floating-point only when S, D, X,
<hexadecimal>, or P used)
<floating-point hex>

DESCRIPTION: DCB directive causes the assembler to allocate a block of bytes, words, or long words, quad words (.D), or hex words (.X or .P) depending upon the <size> specified. If <size> is omitted, word (.W) is the default size. The block length is specified by the absolute expression <length>, which may not contain undefined, forward, or external references. The initial value of each storage unit allocated will be the sign-extended expression <value>, which may contain forward references. <length> must be greater than zero. <value> may be relocatable unless byte size (.B) is specified.

3.4.4 DS - Define Storage

FORMAT:

```
[<label>]  DS.B  <operand>  Define storage in bytes
[<label>]  DS.W  <operand>  Define storage in words (default)
[<label>]  DS.L  <operand>  Define storage in long words
[<label>]  DS.S  <operand>  Define storage in long words
                            (MC68881 only)
[<label>]  DS.D  <operand>  Define storage in quad words
                            (MC68881 only)
[<label>]  DS.X  <operand>  Define storage in hex words
                            (MC68881 only)
[<label>]  DS.P  <operand>  Define storage in hex words
                            (MC68881 only)
```

DESCRIPTION: DS directive is used to reserve memory locations. The contents of the memory reserved are not initialized in any way.

Examples:

```
        DS.B   10       Define 10 contiguous bytes in memory
        DS     10       Define 10 contiguous words in memory
PT1     DS     $10      Define 16 contiguous words in memory
PT2     DS.L   100      Define 100 contiguous long words in memory
        DX.X   10       Define 10 contiguous hex words in memory
```

The label will reference the lowest address of the defined storage area. If word, longword, single, double, extended precision, or packed BCD mode is specified, the storage area is aligned on a word boundary.

```
Example:  DS.B    1     RESERVE ONE BYTE
          DS      0
          DS.W    0     SET LOCATION COUNTER TO EVEN BOUNDARY
          DS.L    0
```

The operand must be absolute and may not contain forward, undefined, or external references.

3.5 LISTING CONTROL AND OUTPUT OPTIONS

3.5.1 FAIL - Programmer Generated Error

FORMAT: FAIL <expression>

DESCRIPTION: The FAIL directive causes an error or warning message to be printed by the assembler. The total error count or warning count is incremented as with any other error or warning. The FAIL directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the error has been printed. The <expression> is evaluated and printed as the error or warning number on the assembly listing. Errors are numbered 0-499; warnings are numbered 500 and above.

3.5.2 FOPT - Floating-Point Assembler Options (MC68020/MC68881 only)

FORMAT: (option>[,<option>] ... [<comment>]

DESCRIPTION: Follows the command format.

OPTIONS: ID = Co-processor identification. Allows more than one MC68881 in a system. New instructions can be defined using existing macro capabilities. An example would be creating two differrent macros:

```
F2ADD.S ...
F3ADD.S ...
```

where the macro definition of F2ADD.S begins with "FOPT ID=2" while F3ADD.S begins with "FOPT ID=3".

The default value for ID is 1.

ROUND=<type> Select IEEE rounding type. Values for <type> are:

N Round to nearest representation (the even value when two numbers exist).

P Round toward plus infinity.

M Round toward minus infinity.

Z Round toward zero; positive numbers are rounded down and negative numbers are rounded up.

PREC=<type> Select IEEE precision type. Values for <type> are:

X Extended precision (default)

D Double precision

S Single precision

3.5.3 FORMAT - Format The Source Listing

FORMAT: FORMAT

DESCRIPTION: Format the source listing, including column alignment (refer to Table 4-1) and structured syntax indentation (refer to paragraph 6.5.4). This option is selected by default.

3.5.4 NOFORMAT - Do Not Format The Source Listing

FORMAT: NOFORMAT

DESCRIPTION: The source listing has the same format as the source input file.

3.5.5 LIST - List The Assembly

FORMAT: LIST

DESCRIPTION: Print the assembly listing on the output device. This option is selected by default. The source text following the LIST directive is printed until an END or NOLIST directive is encountered.

3.5.6 NOLIST - Do Not List The Assembly

FORMAT: NOLIST or NOL

DESCRIPTION: Suppress the printing of the assembly listing until a LIST directive is encountered.

3.5.7 LLEN - Line Length

FORMAT: LLEN n

DESCRIPTION: Set the number of columns to be output to n. The minimum value of n is 72 and the maximum 132. The default value for n is 132 columns.

3.5.8 NOOBJ - No Object

FORMAT: NOOBJ

DESCRIPTION: Suppress the generation of object code.

3.5.9 OPT - Assembler Options

FORMAT: OPT <option>[,<option>]... [<comment>]

DESCRIPTION: Follows the command format.

OPTIONS:

A — Absolute address. All non-indexed operands which reference either labels or the current assembler location counter (*) is resolved as absolute addresses.

NOA — Disable A (default).

BRL — Forward branch long (default). Forward references in relative branch instructions (Bcc, BRA, BSR) will assume the longer form (16-bit displacement, yielding a 4-byte instruction).

A 32-bit displacement is assumed unless the directive "OPT OLD" is in effect (MC68020 only).

BRS or BRB — Forward branch short. As with BRL, but using the shorter form (8-bit displacement, yielding a 2-byte instruction).

BRW — Generate default branch size of 16 bits.

CEX — Print DC expansions.

NOCEX — Opposite of CEX (default).

CL — Print conditional assembly directives (default).

NOCL — Opposite of CL.

CRE — Print cross-reference table at end of source listing. This option must precede first symbol in source program. If this option is not in effect, only the symbol table is printed.

D Debug option (output symbol table to file with the same name as the object code file, but with an extension of ".RS").

EQU Retain equates not used by the program in the symbol table and debug file.

NOEQU Remove unused equates (default).

FRL Forward reference long (default). Forward references in the absolute format assumes absolute long mode (32-bit).

FRS Forward reference short. Forward references in the absolute format assumes absolute short mode (16-bit).

MC Print macro calls (default).

NOMC Opposite of MC.

MD Print macro definitions (default).

NOMD Opposite of MD.

MEX Print macro expansions.

NOMEX Opposite of MEX (default).

O Create output module (default).

NOO Opposite of O.

OLD Interpret the branch size code .L as being a 16-bit branch. Also interpret future uses of "OPT BRL" as referring to forward 16-bit branches.

NOOLD Change back to new branch size meanings for size .L (MC68020 only).

PCO PC relative addressing within ORG. Employ relative addressing, when possible, on backward references occurring in an ORG section.

NOPCO Disable PCO (default).

PCS Force PC relative addressing within SECTION. Forces PC relative addressing (whenever such an addressing mode is legal) in an instruction which occurs within a relocatable SECTION and references an operand in a relocatable SECTION (need not be the same SECTION as the instruction). Failure to resolve such a reference into a 16-bit displacement from the PC results in an error. This option may be used to force position independent code (refer to Chapter 7); however, this option does not force PC relative addressing of absolute operands (defined in ORG section) or unknown forward references.

NOPCS Disable PCS (default).

P=<type> Select microprocessor type; <type> may be 68000, 68010, 68020, or 68881. Default is 68000. Note that P=68881 can be in effect concurrently with P=68000, P=68010, or P=68020. This can be written on a single line, for example by saying OPT P=68010/68881. If P=68010, 68020, or 68881, it must appear before any of the special MC68010, MC68020, or MC68881 instructions, respectively (or it may be specified on the command line; refer to Chapter 4).

3.5.10 PAGE - Top Of Page

FORMAT: PAGE

DESCRIPTION: Advance the paper to the top of the next page. The PAGE directive does not appear on the program listing. No label or operand is used, and no machine code results.

3.5.11 NOPAGE - Do Not Page Source Output

FORMAT: NOPAGE

DESCRIPTION: Suppress paging to the output device. Output lines are printed continuously with no page headings or top and bottom margins.

3.5.12 SPC - Space Between Source Lines

FORMAT: SPC n

DESCRIPTION: Output n blank lines on the assembly listing. This has the same effect as inputting n blank lines in the assembly source. A blank line is defined by the assembler to be a line with only a carriage return.

3.5.13 TTL - Title

FORMAT: TTL <title string>

DESCRIPTION: Print the <title string> at the top of each page. A title consists of up to 60 characters. The same title will appear at the top of all successive pages until another TTL directive is encountered. In order to print a title on the first listing page, the TTL directive must precede the first source line which will appear on the listing.

3.6 LINKAGE EDITOR CONTROL

3.6.1 IDNT - Relocatable Identification Record

FORMAT: <module name> IDNT <version>,<revision> [<descr>]

DESCRIPTION: Every relocatable object module must contain an identification record as a means of identifying the module at link time. The module name is specified in the label field of the IDNT directive, while the version and revision numbers are specified as the first and second operands, respectively. The comment field of the IDNT directive is also passed on to the linkage editor as a description of the module.

3.6.2 XDEF - External Symbol Definition

FORMAT: XDEF <symbol>[,<symbol>]... [<comment>]

DESCRIPTION: This directive specifies symbols defined in the current module that are to be passed on to the linkage editor as symbols which may be referenced by other modules linked to the current module.

3.6.3 XREF - External Symbol Reference

FORMAT: XREF[.S] [<section>:]<symbol> [,<symbol>]...
[,[<section>:]<symbol> [,<symbol>]...]...

DESCRIPTION: This directive specifies symbols referenced in the current module but defined in other modules. This list is passed on to the linkage editor. Each symbol is associated with the specified <section> number which it follows. (Symbols may occur in any section, including an absolute ORG section, if no <section> designation is specified; see following example.)

".S" indicates the XREF symbols will be linked into low address memory so that direct addressing of these symbols may be accomplished through absolute short mode.

EXAMPLE: XREF AA,2:E2,3:E3,B3,C3

The symbol AA can be in any section; E2 is in section 2; and E3, B3, and C3 are in section 3.

CHAPTER 4

INVOKING THE ASSEMBLER

4.1 INTRODUCTION

The flexible, multitask environment of the VERSAdos and the SYSTEM V/68 Operating Systems are similar in Command Line Format, notably in the options supported, and the assembly output file areas. Both systems are discussed below.

4.2 VERSAdos ENVIRONMENT

4.2.1 Command Line Format

The command line format for the assembler running under VERSAdos is:

```
ASM   <source file>[,[<object file>][,<listing file>]][;<options>]
```

Only the <source file> is required. The default extension on the <source file> is SA. If the <object file> and/or <listing file> are not specified, they will default to the same filename as the <source file>, but with extensions of RO and LS, respectively. The following command lines are equivalent:

```
ASM   TEXT
ASM   TEXT,TEXT,TEXT
ASM   TEXT.SA,TEXT.RO,TEXT.LS
```

NOTES

1. The source file exists on a device which supports VERSAdos Block I/O. For example, the source file cannot be the user's console (#).

2. #NULL is not allowed as an object file. Users who wish to inhibit the generation of an object file should specify the command line option -C.

Default extensions are assumed for <object file> and <listing file>, if not specified. Multiple source files may be assembled by separating these input files with a slash (/). In the case of multiple source files, the first file name is used for the default object and listing filenames. The listing may be output to the CRT or the printer during assembly by specifying the appropriate mnemonic in place of the listing file; e.g., the command ASM TEXT,,#PR prints the listing.

The assembler recognizes the following options on the command line:

C	Produce object code (default).
-C	Inhibit production of object code.
D	Produce symbolic debug symbol table file.
-D	Inhibit production of debug file (default).
F	Enable floating-point warning messages during assembly (default) (MC68881 only).
-F	Disable floating-point warning messages during assembly (MC68881 only).
L	Produce listing (default).
-L	Inhibit listing.
M	List macro expansions.
-M	Inhibit listing of macro expansions.
O	Branch size code extensions are the same as in previous M68000 assemblers.
-O	Offers same functionality as directive "OPT NOOLD" (MC68020 only).
P=68000	Accept MC68000 instruction set (default).
P=68010	Accept MC68010 instruction set.
P=68020	Accept MC68020/MC68881 instruction set.
P=68xxx/68881	Accept MC68881 instruction set, where xxx is 000, 010 or 020.
R	Produce cross-reference.
-R	Inhibit production of cross-reference (default).
S	List structured control expansions.
-S	Inhibit listing of structured control expansions (default).
W	Enable warning messages during assembly (default).
-W	Disable warning messages during assembly.
Z=<size>	Increase data area size (default is 37K).

Multiple options are typed without separation -- e.g., ;LM-CP=68000. Refer also to paragraph 3.5.2.10 for assembler options which may be included in the source code with the OPT directive. When there is a conflict between an option specified on the command line and one specified with the OPT directive, the command line option overrides.

4.2.2 Symbol Table Size Option

The symbol table size may be increased by specifying the Z option:

```
Z=<size>
```

where:

<size> is the number of Kbytes to be used in the data (stack + heap) area of the assembler. <size> is in K (1024) bytes.

For example:

```
ASM   TEST,,#PR;RZ=40
```

will assemble the source program in TEST.SA, put the relocatable code in TEST.RO, and send the listing, including cross-references, to the printer rather than to a listing file. The data area will be 40K bytes.

4.2.3 Microprocessor Type Option

The microprocessor type can be specified with the P=<type> option on the command line, where <type> may be 68000, 68010, 68020, or 68881. If omitted, default is P=68000.

NOTE

The MC68881 floating-point co-processor can be specified with any of the previous values by separating the two values with a slash (/) --e.g., P=68020/68881.

4.3 SYSTEM V/68 ENVIRONMENT

4.3.1 Command Line Format

The command line format for the assembler running under SYSTEM V/68 is:

asm [<sep><option>] ... <sep><source file>

where:

asm	is the SYSTEM V/68 command that invokes the assembler.
<sep>	is a field separator consisting of one or more spaces.
<source file>	is the single input file for the assembler (file lists are not supported for SYSTEM V/68).
<option>	is any option that may be accepted by the assembler. More than one option may be specified. Options may be specified after <source file> as well as before it.

The syntax of <option> is:

+|- option char>[<option param>]

where:

+	is option enable (required, unlike VERSAdos)
-	is option disable
<option char>	is a one character option identifier in equivalent upper or lowercase characters defined exactly the same as in the VERSAdos environment.
<option param>	is any required/allowed parameter which immediately follows the option character, e.g., = 68010 or = <size>.

The SYSTEM V/68 options supported are the same as the options for a VERSAdos environment with the exception of the c (or C), l (or L) and d (or D) options. The +c option allows an option parameter. This option parameter specifies the filename of the object code file. When +c appears without an option parameter, a default filename based on the <source file> name is generated. Similarly, the +l and +d option allows an option parameter that specifies the filename of the listing file and symbol table file, respectively. The +l and +d options provide for a similar default file naming convention.

As in the VERSAdos environment, the user may allow the assembler output file name to be defaulted. If a listing file or an output file is generated and the user has not specified the name of the listing or output file, the filename is based on the <source file> name. If the <source file> name is a pathname, the pathname is stripped from the filename, so that the listing file or output file name resides in the current working directory.

A suffix is expected for all files (source, listing, and output). The suffix is defined as a "." followed by zero or more characters. When no suffix exists, a default suffix is appended by the assembler.

The default suffixes (even for uppercase filenames) are:

.sa	source file
.ls	listing file
.ro	object code file
.rs	symbol table file

If the +l option appears without an option parameter, the listing filename is the source filename with its suffix replaced by ".ls". If the +l option appears without a suffixed option parameter, the ".ls" suffix is added. Similar defaulting occurs for the +c and +d options.

Due to difficulties with syntax, the file list supported in the VERSAdos environment is not supported in the SYSTEM V/68 environment. Only one input file can be named on the command line of the assembler. To add some relief to this restriction, INCLUDE directive files may be nested one level.

4.4 ASSEMBLER OUTPUT

Assembler output includes an assembly listing, a symbol table, a symbolic debug symbol table file, and an object program file.

The assembly listing includes the source program, as well as additional information generated by the assembler. Most lines in the listing correspond directly to a source statement. Lines which do not correspond directly to a source line include:

- Page header and title
- Error and warning lines
- Expansion lines for instructions over three words in length

The assembly listing format is shown in Table 4-1. The label, operation, and operand fields may be extended if the source field does not fit into the designated output field.

The last page of the assembly listing is the symbol table. Symbols are listed in alphabetical order, along with their values and an indication of the relocatable section in which they occur (if any). Symbols that are XDEF, XREF, REG, in named common, or multiply defined are flagged. If option CRE has been specified in the program, the cross-reference listing will identify the source lines on which the symbol was defined or referenced (definitions appear first, flagged with a "-").

An example of assembler output is provided in Appendix C.

If the option "D" was specified either in the source program or on the command line, the symbolic debug symbol table is output to a file given the same name as the relocatable object file, with an extension of ".RS". Linking (with the linker's "D" option) makes this information available for easy debugging with the SYMbug program. Refer to the MC68000 Family Linkage Editor User's Manual, Appendix D, for .RS file formats or the SYSTEM V/68 PAL Linkage Editor User's Manual, Appendix E.

TABLE 4-1. Standard Listing Format

COLUMNS	CONTENTS	EXPLANATION
1-4	Source line number	4-digit decimal counter
6	Section number	1-digit hex section number (blank indicates location counter is absolute)
8-15	Location counter value	In hex
17-20	Operation word	In hex
21-24	First extension word	In hex
25-28	Second extension word	In hex; any additional extension words appear on the next line
30-37	Label field	
39-46	Operation field	
48-67	Operand field	
70-N	Comment field	

4.5 ASSEMBLER RUNTIME ERRORS

During runtime, the assembler may generate its own error messages. These are listed in Appendix E. However, since the assembler is a Pascal program and operates in the VERSAdos operating system environment or the SYSTEM V/68 environment, runtime errors may occur from these sources as well. Refer to the VERSAdos Messages Reference Manual or the SYSTEM V/68 Pascal Compiler User's Manual for applicable runtime error messages.

Any assembly instruction generating six or more bytes of code, which is found to have an operand error, can generate six bytes of object code. The code for the instruction is $4AFB, which is an illegal opcode; the extension word(s) is $4E71, which is a NOP. These six bytes allow more instructions to be patched in place or a jump to be inserted to a patch area anywhere in the address space.

Instructions which generate only two or four bytes continue to generate a 2- or 4-byte length instruction, respectively, whenever an operand is in error. The instruction word, however, is illegal, and the extension is a NOP.

Undefined operations generate six bytes of code with an illegal opcode and NOP extensions.

CHAPTER 5

MACRO OPERATIONS AND CONDITIONAL ASSEMBLY

5.1 INTRODUCTION

This chapter describes the macro (paragraph 5.2) and the conditional assembly (paragraph 5.3) capabilities of the assembler. These features can be used in any program.

5.2 MACRO OPERATIONS

Programming applications frequently involve the coding of a repeated pattern of instructions that, within themselves, contain variable entries at each iteration of the pattern, or basic coding patterns subject to conditional assembly at each occurrence. In either case, macros provide a shorthand notation for handling these patterns. Having determined the iterated pattern, the programmer can, within the macro, designate fields of any statement as variable. Thereafter, by invoking a macro, the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

Macro usage can be divided into two basic parts -- definition and expansion.

When the pattern is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently invoked (called). The name of a macro definition should not be the same as an existing instruction mnemonic or an assembler directive.

Expansion occurs when the previously defined macro is called (invoked). The macro call causes source statements to be generated. The generated statements may contain substitutable arguments. The statements that may be generated by a macro call are relatively unrestricted as to type. They can be any processor instruction, almost any assembler directive, or any previously-defined macro. Source statements generated by a macro call are subject to the same conditions and restrictions to which programmer-generated statements are subject.

The invocation of a macro requires that the macro name appear in the operation field of a source statement. Most arguments are placed in the operand field. Appropriate arguments selected according to the macro definition cause the assembler to produce in-line coding variations of the macro definition.

The effect of a macro call is the same as an open subroutine in that it produces in-line code to perform a predefined function. The in-line code is inserted in the normal flow of the program so that the generated instructions are executed in-line with the rest of the program each time the macro is called.

5.2.1 Macro Definition

The definition of a macro consists of three parts:

a. The header: <label> MACRO

   The <label> of the MACRO statement is the "name" by which the macro is later invoked. This name must be a unique class 1 symbol. A macro name may not have a period (.) as any character other than the first.

b. The body

   The body of a macro is a sequence of standard source statements. Macro parameters are defined by the appearance of argument designators within these source statements. Legal macro-generated statements include the set of MC68000, MC68010, MC68020, and MC68881 assembly language instructions, assembler directives, structured syntax statements, and calls to other, previously defined macros. However, macro definitions may not be nested. When macro text lines are saved for later expansion, all spaces in the source line are compressed. This space compression will be noticed only if the listing is unformatted or if the macro text includes literal strings with multiple spaces (which would not expand correctly). Macro expansion lines which contain more than 80 characters are truncated at 80 characters, which is the maximum length of an assembler input line.

c. The terminator: ENDM

5.2.2 Macro Invocation

The form of a macro call is: [<label>] <name>[.<qualifier>] [<parameter list>]

Although a macro may be referenced by another macro prior to its definition in the source module, the macro must be defined before its first in-line expansion. The name of the called macro must appear in the operation field of the source statement; parameters may appear as qualifiers to the macro name and/or in the operand field of the source statement, separated by commas.

The macro call produces in-line code at the location of the invocation, according to the macro definition and the parameters specified in the macro call. The source statements so generated are then assembled, subject to the same conditions and restrictions affecting any source statement. Nested macro calls are also expanded at this time.

5.2.3 Macro Parameter Definition and Use

Up to 36 different, substitutable arguments may appear in the source statements which constitute the body of a macro. These arguments are replaced by the corresponding parameters in a subsequent call to that macro.

Arguments are designated by a backslash character (\), followed by a digit (0 through 9) or an uppercase letter (A through Z). Argument designator \0 refers to the qualifier appended to the macro name; parameters in the operand field of the macro call refer to argument designations \1 through \9 and \A through \Z, in that order.

The parameter list (operand field) of a macro call may be extended onto additional lines if necessary. The line to be extended must end with a comma separating two parameters, and the subsequent extension line must begin with an ampersand (&) in column 1. The extension of the parameter list will begin with the first non-blank characters following the ampersand. No other source lines may occur within an extended parameter call, and no comment field may occur except after the last parameter on the last extension line.

Argument substitution at the time of a macro call is handled as a literal (string) substitution. The string corresponding to a given parameter is substituted literally wherever that argument designator occurs in a source statement as the macro is expanded. Each statement generated in this expansion is assembled in-line. (Note that, if a qualifier is present, argument \0 begins with the first character following the period which separates the qualifier from the macro name.)

It is possible to specify a null argument in a macro call by an empty string (not a blank); except for \0, it must still be separated from other parameters by a comma. In the case of a null argument referenced as a size code, the default size code (W) is implied; when a null argument itself is passed as an argument in a nested macro call, a null argument is passed. All parameters have a default value of null at the time of a macro call.

If an argument has multiple parts or contains commas or blanks, the entire argument must be enclosed within angle brackets (< and >) as required characters. Such arguments must still be separated from other arguments by commas. A bracketed argument with no intervening character is treated as a null argument. Embedded brackets must occur in pairs. Parameter \0 may not be bracketed and, hence, may not contain blanks (although commas are legal). Note that a macro argument may not contain the characters "<" or ">" unless they occur as part of the argument bracketing.

5.2.4 Labels Within Macros

To avoid the problem of multiply defined labels resulting from multiple calls to a macro which employs labels in its source statements, the programmer may direct the assembler to generate unique labels on each call to a macro.

Assembler-generated labels include a string of the form .nnn, where nnn is a 3-digit value. The programmer may request an assembler-generated label by specifying \@ in a label field within a macro body. Each successive label definition which specifies a \@ directive will generate successive values of .nnn, thereby creating unique labels on repeated macro calls. Note that \@ may be preceded or succeeded by additional characters for clarity and to prevent ambiguity (more than four preceding characters may introduce a problem with non-uniqueness of symbols).

References to an assembler-generated label always refer to the label of the given form defined in the current level of macro expansion. Such a label is referenced as an operand by specifying the same character string as that which defines the label.

5.2.5 The MEXIT Directive

The MEXIT directive terminates the macro source statement generation during expansion. It may be used within a conditional assembly structure (refer to paragraph 5.3) to skip any remaining source lines up to the ENDM directive. All conditional assembly structures pending within the macro currently being expanded are also terminated by the MEXIT directive.

Example:

```
SAV2    MACRO
        MOVE.L    \1,SAVET        SAVE 1ST ARGUMENT
        MOVE.L    \2,SAVET+4      SAVE 2ND ARGUMENT
        IFC       '\3',''         IS THERE A 3RD ARGUMENT?
        FAIL      1000            DID ASSEMBLER GO THRU HERE?
        MEXIT                     NO, EXIT FROM MACRO
        ENDC
        MOVE.L    \3,SAVET+8      SAVE 3RD ARGUMENT
        ENDM
```

5.2.6 NARG Symbol

The symbol NARG is a special symbol when referenced within a macro expansion. The value assigned to NARG is the index of the last argument passed to the macros in the parameter list (even if nulls). NARG is undefined outside of macro expansion and may be referenced as a Class 1 or 2 user-defined symbol outside of a macro expansion.

5.2.7 Implementation of Macro Definition

When the sequence of source statements

```
          .
          .
          .
MAC1      MACRO
          <stmt1>
          <stmt2>
          .
          .
          .
          <stmtn>
          ENDM
          .
          .
          .
```

is encountered in a source program, the following actions are performed:

a. The symbol table is checked for a Class 1 symbol entry of 'MAC1'. If such an entry is already present, a redefined symbol error (231) is generated; if no such entry exists, an entry is placed in the symbol table, identifying MAC1 as a macro.

b. Starting with the line following the MACRO directive, each line of the macro body is saved in a character sequence identified with MAC1. In the example, stmt1 through stmtn are saved in this manner. No object code is produced at this time. A check is made for missing parameter references in the macro text (e.g., parameters \1, \2, and \4 are referenced, but \3 is not).

c. Normal processing resumes with the line following the ENDM directive.

5.2.8 Implementation of Macro Expansion

When the statement:

MAC1.<qualifier> <param1>,<param2>,...,<paramn>

is encountered in a source program calling the previously defined macro MAC1 (above), the following actions are performed:

a. Because the label field is blank, the string MAC1 is recognized as the operation code of the instruction. The symbol table is consulted for a Class 1 symbol entry with this name. If no such entry exists, an undefined symbol error (238) is generated. In this case, the entry indicates that the symbol identifies a macro.

b. The rest of the line is scanned for parameters which are saved as literals or null values, one such value in each of the 36 parameter record fields. If the source line ends with a comma, the next line is checked for an extension of the parameter list. A cross-check is made with the macro definition for the number of parameters in the call. No object code is produced.

c. Macro expansion consists of the retrieval of the source lines which comprise the macro body. Each line is retrieved in turn, with special character pairs replaced by parameter strings or assembler-generated label strings.

   If a backslash character (\) is followed by either a digit (0 through 9) or an uppercase letter (A through Z), the two characters are replaced by the literal string which corresponds to that parameter on the macro invocation line(s).

   A character sequence which includes \@ is replaced by an assembler-generated label, as defined in paragraph 5.2.4. An assembler-generated label is uniquely identified by the characters preceding and/or appended to the \@ sequence and the macro invocation in which the reference occurs. Such labels may appear anywhere in the source line and always refer to the current macro expansion.

   NOTE

   Space compression is automatically done within macros. For example, the instruction DC.B ' ' becomes DC.B ' '.

d. When a line has been completely expanded, it is assembled as any other source input line. At this time, any errors in the syntax of the expanded assembly code are found. Expanded lines longer than 80 characters are truncated, and an error code is generated.

   If a nested macro call is encountered, the nested macro expansion takes place recursively. There is no set limit to the depth of macro call nesting.

5.3 CONDITIONAL ASSEMBLY

Conditional assembly allows the programmer to write a comprehensive source program that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros and through definition of symbols via the SET and EQU directives. Variations of parameters can then cause assembly of only those parts necessary for the specified conditions.

The I/O section of a program, for example, will vary, depending on whether the program is used in a disk environment or in a paper tape environment. Conditional assembly directives can include or exclude an I/O section, based on a flag set at the beginning of the assembly.

5.3.1 Conditional Assembly Structure

The conditional assembly structure consists of three parts:

a. The header

There are two conditional clauses recognized by the assembler. The first form compares the equality of two strings:

```
IFxx      <string1>,<string2>
```

"xx" specifies either the string compare (C) condition or the string not compare (NC) condition, representing string equality and inequality, respectively. The result of the string comparison, along with the "xx" condition, determines whether the body of the conditional structure will be assembled. Either string may contain embedded commas or spaces. An apostrophe that occurs within a string must be specified by double apostrophes.

The second form of the conditional clause compares an expression against zero:

```
IFxx      <expression>
```

"xx" specifies a conditional relation between the expression and the value zero. The result of this comparison at assembly time determines whether the body of the conditional structure will be assembled. Valid conditional relation codes include:

```
EQ:  expression =  0
NE:  expression <> 0
LT:  expression <  0
LE:  expression <= 0
GT:  expression >  0
GE:  expression >= 0
```

Because of the nature of this comparison, the expression must be absolute. No forward references are allowed.

b. The body

The body of the conditional assembly structure consists of a sequence of standard source statements. There is no set limit to the depth of conditional assembly nesting; if such nesting occurs, a terminator must be specified for each structure.

c. The terminator: ENDC

When an IFxx directive is encountered, the specified condition is evaluated. If the condition is true, the statements constituting the body of the conditional assembly structure are each assembled in turn. If the relation is false, the entire conditional assembly structure is ignored; the ignored lines are not included in the assembly listing. By specifying the OPT NOCL option (paragraph 3.5.2.10), the header and terminator lines are ignored for listing purposes.

IFxx and ENDC directives may not be labeled.

Testing for null parameters may be done via the string compare form of the conditional assembly. To assemble conditionally if parameter 1 is null, either of the following directives is correct:

```
IFxx  '','\1'
   or
IFxx  '\1',''
```

where:

xx = C or NC

To assemble conditionally if a parameter is present, use either of the IFNC formats analogous to the above two.

A conditional assembly structure is also terminated by a MEXIT directive, as explained in paragraph 5.2.5. All conditional assembly structures which originate in a macro are terminated at the exit from that macro (if not before). Only conditional assembly structures which originated within a given macro may be terminated within that macro. These two rules are necessary for the consistent implementation of conditional assembly.

5.3.2 Example of Macro and Conditional Assembly Usage

The following example illustrates most of the features of macros and conditional assembly structures. The assembly code is shown as it appears, without line numbers or object code. Note that angle brackets (< >) shown in examples are required characters.

```
          .
          .
          .
MACO      MACRO
          MOVE.\0        \1
          CLR.L          \2
          ENDM
          .
          .
          .
```

```
MAC1        MACRO
            MOVE.\0         #\1,D\2
            IF\3            \1                      CONDITIONAL
            ADD.\0          #1,D\2
            IF\3            \1-5                    NESTED CONDITIONAL
            ADD.\0          #2,D\2                  \4
            ENDC                                    END NESTED CONDITIONAL
            ENDC                                    END CONDITIONAL
LAB\@       CLR.L           D1
            MOVE.\0         D\2,(A0)+
            B\3             \@END
            BRA             LAB\@
\@END       \5.\0           #1,D\2
            IFLE            \1
            MACO.\0         <D\2,(A0)>,A\2          NESTED MACRO CALL
            ENDC
            ENDM
            .
            .
            .
            OPT             MEX,NOCL
            MAC1.L          7,3,GT,<TEST PASSES>,ADD
            MOVE.L          #7,D3
            ADD.L           #1,D3
            ADD.L           #2,D3                   TEST PASSES
LAB.001     CLR.L           D1
            MOVE.L          D3,(A0)+
            BGT             .002END
            BRA             LAB.001
.002END     ADD.L           #1,D3
            .
            .
            .
            MAC1            0,6,NE,<ERROR HERE>,SUB
            MOVE.           #0,D6
LAB.003     CLR.L           D1
            MOVE.           D6,(A0)+
            BNE             .004END
            BRA             LAB.003
.004END     SUB.            #1,D6
            MACO.           <D6,(A0)>,A6            NESTED MACRO CALL
            MOVE.           D6,(A0)
            CLR.L           A6
            .
            .
            .
```

CHAPTER 6

STRUCTURED CONTROL STATEMENTS

6.1 INTRODUCTION

An assembly language provides an instruction set for performing certain rudimentary operations. These operations, in turn, may be combined into control structures -- such as loops (for, repeat, while) or conditional branches (if-then, if-then-else). The assembler, however, accepts formal, high-level directives that specify these control structures, generating, in turn, the appropriate assembly language instructions for their efficient implementation. This use of structured control statement directives improves the readability of assembly language programs, without compromising the desirable aspects of programming in an assembly language.

6.2 KEYWORD SYMBOLS

The following Class 1 symbols, used in the structured syntax, are reserved keywords (directives):

ELSE	ENDW	REPEAT
ENDF	FOR	UNTIL
ENDI	IF	WHILE

The following symbols are required in the structured syntax, but are nonreserved keywords:

AND	DOWNTO	TO
BY	OR	
DO	THEN	

Note that AND and OR are reserved instruction mnemonics, however.

6.3 SYNTAX

The formats for the IF, FOR, REPEAT, and WHILE statements are found in paragraphs 6.3.1 through 6.3.4. They are spaced to show the line separations required for Class 1 symbol usage (paragraph 6.5.1). Syntactic variables used in the formats are as follows:

<expression> A simple or compound expression (paragraph 6.4).

<stmtlist> Zero or more assembler directives, structured control statements, or executable instructions.

Note that an assembler directive (Chapter 3) occurring within a structured control statement is examined exactly once - at assembly time. Thus, the presence of a directive within a FOR, REPEAT, or WHILE statement does not imply repeated occurrence of an assembler directive; nor does the presence of a directive within an IF-THEN-ELSE statement imply a conditional assembly structure (Chapter 5).

For correct recognition, the statements in <stmtlist> must not appear on the same line as the structured syntax symbols.

<size> The value B, W, or L, indicating a data size of byte, word, or longword, respectively. With the keyword FOR, <size> is a single code applying to <op1>, <op2>, <op3>, and <op4>. With the keywords IF, UNTIL, and WHILE, <size> indicates the size of the operand comparison in the subsequent simple expression (refer to paragraph 6.4.2 for a compound expression). Note that structured syntax statements rely on the underlying opcodes and the restrictions these opcodes place on arguments to the statements. For example, the structured syntax statement

```
FOR.B D7 = #0 to #255 DO
```

generates code without warning but does not execute as expected. This is because the comparison opcode CMP does a signed comparison and hence deals with numbers in the range -128...127 instead of 0...255. (MC68881 only: only IF is now implemented with floating-point ranges.)

<extent> The value S or L, indicating that the branch extent is short or long, respectively. This is appended to the keywords THEN, ELSE, and DO, to force the appropriate extent of the forward branch over the subsequent <stmtlist>. The default extent for the MC68020 is determined by the option directive (OPT, BRS, OPT BRB, OPT BRW, or OPT BRL) currently in effect.

<op1> A user-defined operand whose memory/register location holds the FOR-counter. The effective address must be an alterable mode.

<op2> The initial value of the FOR-counter. The effective address may be any mode.

<op3> The terminating value for the FOR-counter. The effective address may be any mode.

<op4> The step (increment/decrement) for the FOR-counter each time through the loop. If not specified, it defaults to a value of #1. The effective address may be any mode.

6.3.1 IF Statement

SYNTAX:

```
IF[.<size>]  <expression>  THEN[.<extent>]
  <stmtlist>
ENDI
```

or

```
IF[.<size>]  <expression>  THEN[.<extent>]
  <stmtlist>
ELSE[.<extent>]
  <stmtlist>
ENDI
```

FUNCTION: If <expression> is true, execute <stmtlist> following THEN; if <expression> is false, execute <stmtlist> following ELSE, if present, or advance to next instruction.

NOTES:

a. If an operand comparison <expression> is specified, the condition codes are set and tested before execution of <stmtlist>.

b. In the case of nested IF-THEN-ELSE statements, each ELSE refers to the closest IF-THEN.

6.3.2 FOR Statement

SYNTAX:

```
FOR[.<size>] <op1> = <op2>  TO  <op3>  [BY <op4>]  DO[.<extent>]
  <stmtlist>
ENDF
```

or

```
FOR[.<size>] <op1> = <op2>  DOWNTO  <op3>  [BY <op4>]  DO[.extent>]
  <stmtlist>
ENDF
```

FUNCTION: These counting loops utilize a user-defined operand, <op1>, for the loop counter. FOR-TO allows counting upward, while FOR-DOWNTO allows counting downward. In both loops, the user may specify the step size, <op4>, or elect the default step size of #1. The FOR-TO loop is not executed if <op2> is greater than <op3> upon entry. Similarly, the FOR-DOWNTO loop is not executed if <op2> is less than <op3>.

NOTES:

a. The condition codes are set and tested before each execution of <stmtlist>. This happens even if <stmtlist> is not executed.

b. A step size of #1 may not be meaningful if the counter, <op1>, is used to index through word or longword-sized data.

c. Each immediate operand must be preceded by a # sign. For example, the following would loop ten times by steps of four.

```
FOR  COUNT = #4  TO #40  BY #4  DO ...
```

d. The FOR structure generates a move, a compare, and either an add or subtract. Therefore, if any of the four operands is an A register, <size> may not be B (byte).

6.3.3 REPEAT Statement

SYNTAX:

```
REPEAT
  <stmtlist>
UNTIL[.<size>]  <expression>
```

FUNCTION: <stmtlist> is executed repeatedly until <expression> is true.

NOTES:

a. The <stmtlist> is executed at least once, even if <expression> is true upon entry.

b. If an operand comparison <expression> is specified, the condition codes are set and tested following each execution of <stmtlist>.

6.3.4 WHILE Statement

SYNTAX:

```
WHILE[.<size>]  <expression>  DO[.<extent>]
  <stmtlist>
ENDW
```

FUNCTION: The <expression> is tested before execution of <stmtlist>. While <expression> is true, <stmtlist> is executed repeatedly.

NOTES:

a. If <expression> is false upon entry, <stmtlist> is not executed.

b. If an operand comparison <expression> is specified, the condition codes are set and tested before each execution of <stmtlist>. The condition codes are set and tested even if <stmtlist> is not executed.

6.3.5 (MC68020/MC68881 only.) Floating-Point Structured Assembler Syntax

```
IF FPn <Ffpcc> <ea> THEN
   ...

IF <ea> <Ffpcc> FPn THEN
   ...

IF FPn <Ffpcc> FPm THEN
   ...

IF <Ffpcc> THEN
```

where:

FPm, FPn are floating point registers; Ffpcc is a floating-point condition code, defined in 2.10.25; F is a required constant.

When the assembler expands the structured IF statement with a floating-point condition code, fpcc, it must choose the true IEEE inverse of cc. For example, the code for

```
IF.X  FP3 <FGT> #3.3 THEN        (where GT is one value of fpcc and F is
                                  a required constant value)
```

would be

```
    FCMP.X  #3.3,FP3
    FBNGT   ELSECLAUSE
    ....                          main clause code
    BRA     PAST
ELSECLAUSE
    ....                          else clause code
PAST
    ....
```

NOTE: The branch following the FCMP is a FBNGT rather than a FBLE. FBNGT is the IEEE inverse of FBGT.

6.4 SIMPLE AND COMPOUND EXPRESSIONS

Expressions are an integral part of IF, REPEAT, and WHILE statements. An expression may be simple or compound. A compound expression consists of no more than two simple expressions joined by AND or OR.

6.4.1 Simple Expressions

Simple expressions are concerned with the bits of the Condition Code Register (CCR). These expressions are of two types. The first type merely tests conditions currently specified by the contents of the CCR (paragraph 6.4.1.1). The second type sets up a comparison of two operands to set the condition codes, and afterwards tests the codes (paragraph 6.4.1.2).

6.4.1.1 Condition Code Expressions. Fourteen tests (identical to those in the Bcc instruction) may be performed, based on the CCR condition codes. The condition codes, in this case, are preset by either a user-generated instruction or a structured operand-comparison expression (paragraph 6.4.1.2). Each test is expressed in the structured control statement by a mnemonic enclosed in angle brackets (< >) as required characters, as follows:

```
<CC>
<CS>
<EQ>
<GE>
<GT>
<HI>      For an explanation of each test, see Table A-2,
<LE>      "Conditional Tests", in the MC68000 16-Bit
<LS>      Microprocessor User's Manual.
<LT>
<MI>
<NE>
<PL>
<VC>
<VS>
```

For example:

```
IF      <EQ>  THEN
  CLR.L    D2
ENDI

REPEAT
  SUB      D4,D3
UNTIL   <LT>
```

6.4.1.2 Operand Comparison Expressions. Two operands may be compared in a simple expression, with subsequent transfer of control based on that comparison. Such a comparison takes the form:

<op1> <cc> <op2>

where:

<cc> is a condition mnemonic enclosed in angle brackets (as described in paragraph 6.4.1.1), specifying the relation to be tested between <op1> and <op2>. When processed by the assembler, this expression translates to a compare instruction.

For example:

CMP <op1>,<op2>

followed by a branch instruction (Bcc) which tests the relation specified. <op1> is normally, but not necessarily assigned to the first (leftmost) operand and <op2> to the second (rightmost) operand of the compare instruction.

NOTE

A blank (#' ') should not be used for the value of <op1> or <op2>.

A size may be specified for the comparison by appending a data size code (B, W, or L) to the directive, with W being the default. The only restriction is that a byte-size code (B) may not be used in conjunction with an address register direct operand.

Compare instructions require certain effective addressing modes for their operands. These modes are listed in Table 6-1. However, if the operands, <op1> and <op2>, are not listed in an order that generates a legal compare instruction (Table 6-1), but generates a legal compare if the operand order is reversed, the assembler reverses the operands when expanding the expression. To maintain the nature of the relation specified, the condition operator is adjusted, if necessary. For example, "D2 <GT> #5" is adjusted by the assembler to the equivalent of "#5 <LT> D2"; likewise, "A2 <EQ> (A5)" is adjusted to the equivalent of "(A5) <EQ> A2". This processing allows the user the flexibility of specifying the more meaningful operand order in the expression.

TABLE 6-1. Effective Addressing Modes for Compare Instructions

COMPARE INSTRUCTIONS	EFFECTIVE ADDRESSING MODES FOR: FIRST OPERAND	SECOND OPERAND
CMP	(All)	Data register direct
CMPA	(All)	Address register direct
CMPI	Immediate	(Data alterable)
CMPM	Postincrement register indirect	Postincrement register indirect

If the operands, either as stated or reversed, do not yield a legal compare instruction, an error will result. For example, the statement

```
IF      (A1) <NE> (A2)  THEN
```

results in an ERROR 213 message (illegal address mode) during expansion. To avoid this error, a MOVE is required to accomplish a legal operand, such as:

```
MOVE    (A2),D2
IF      (A1) <NE> D2  THEN
```

Examples:

```
WHILE.B  (A3) <NE> D2  DO          THIS EXPRESSION IS LEGAL AS STATED.
  MOVE.B   (A5)+,D2
ENDW

IF       D7 <LT> #10  THEN         THIS EXPRESSION IS REVERSED.
  BSR      SUBR1
ELSE
  MULS     #2,D7
ENDI
```

6.4.2 Compound Expressions

A compound expression consists of two simple expressions (paragraph 6.4.1) joined by a logical operator. The Boolean value of the compound expression is determined by the Boolean values of the simple expressions and the nature of the logical operator (AND or OR).

The two simple expressions are evaluated in the order in which they are given. However, if an AND separates the expressions and the first expression is false, the second expression is not evaluated. Likewise, if an OR separates the expressions and the first expression is true, the second expression is not evaluated. In these cases, the compound expression is either false or true, respectively, and the condition codes reflect the result of only the first simple expression.

A size may be specified for each operand comparison expression. The size of the comparison for the first expression may be appended to the directive, while the size of the comparison for the second expression may be appended to the keyword AND or OR. For example, in the statement

```
IF.L    D3 <GT> (A0)  OR.B  #'Q' <EQ> BUFFER1
```

the first comparison is a longword comparison, and the second is a byte comparison.

6.5 SOURCE LINE FORMATTING

The format of structured source statements is more restricted than the format of basic statements. The following paragraphs discuss the formatting requirements of structured statements as well as their appearance in the assembly listing.

6.5.1 Class 1 Symbol Usage

Class 1 symbols, as described in paragraphs 2.8.2 and 6.2, are the assembler directives (including macro names), instruction mnemonics, and the structured control directives. Only one of these is recognized on each source line. Thus, each directive (reserved keyword) of a structured control statement and each executable instruction generated by the programmer must be written on a separate source line. The following source line, for example, is in error:

```
REPEAT   MOVE -(A5),D2  UNTIL <EQ>
```

because the MOVE and UNTIL symbols and their operands are not recognized, but are treated as part of the comment field of the REPEAT directive. Likewise, the following lines are in error:

```
IF       <VS>  THEN  JSR OVERFLOW
ELSE     JMP (A3)  ENDI
```

because the JSR, JMP, and ENDI symbols and their operands are not recognized. The correct format for these lines is as follows:

```
REPEAT
  MOVE     -(A5),D2
UNTIL    <EQ>
```

and

```
IF       <VS>  THEN
  JSR      OVERFLOW
ELSE
  JMP      (A3)
ENDI
```

6.5.2 Limited Free-Formatting

To improve readability, limited free-formatting allows the operand field of the IF, UNTIL, WHILE, and FOR directives to be extended onto additional consecutive lines.

For example:

```
IF       #15  <LT>  D7
           AND
         (A3) <NE> D3  THEN


UNTIL    (A7)+ <EQ> D2  OR
               <VS>


FOR      D1 = #1  TO #5
              BY #1  DO
```

6.5.3 Nesting of Structured Statements

Structured statements may be nested as desired to create multilevel control structures. An example of such nesting is the following:

```
IF       <EQ>   THEN
  REPEAT
    MOVE     D0,(A5)+
    ADDQ     #4,D0
    MOVE.L   A4,(A4)+
  UNTIL.L  A5 <LE> A4
ELSE.L
  FOR      D2 = #10  TO #20  BY #2  DO
    WHILE    D4 <LT> D2   AND   D4 <LT> #100   DO
      MOVE.L   10(A3,D4.W),(A5)+
      ADDQ     #2,D4
    ENDW
  ENDF
ENDI
```

6.5.4 Assembly Listing Format

By default (FORMAT directive), the assembly listings are formatted according to Table 4-1. In addition, the operation and operand fields of source lines in structured syntax are indented two columns for each nested level of operation. This automatic formatting may be turned off by using the NOFORMAT directive.

The assembly language code generated for the structured syntax is included in the listing when the S (or s) option is specified in the ASM (or asm) command line.

6.6 EFFECTS ON THE USER'S ENVIRONMENT

If the S (or s) option is specified in the ASM command line (paragraph 4.2.1), the generated code of the structured control expansions is listed. There may be three items found in this code that will affect the user's environment:

a. During assembly, local labels beginning with "Z_L" are generated. These labels use the same increment counter (.nnn) as local labels in macros (paragraph 5.2.4). They are stored in the symbol table and should not be duplicated in user-defined labels.

b. In the FOR loop, <opl> is a user-defined symbol. When exiting the loop, the memory/register assigned to this symbol contains the value which caused the exit from the loop.

c. Compare instructions (Table 6-1) are generated by the assembler whenever two operands are tested relationally in a structured statement. During runtime, however, these assembler-generated instructions set the condition codes of the CCR (in the case of a loop, the condition codes are set repeatedly). Any user-written code, either within or following a structured statement, that references the CCR should be attentive to the effect of these instructions.

CHAPTER 7

GENERATING POSITION INDEPENDENT CODE

7.1 FORCING POSITION INDEPENDENCE

When creating a relocatable program module, it is often desirable to ensure that all references to operands in relocatable sections are position-independent effective addresses -- i.e., no absolute addresses occur as effective addresses for such references. To avoid absolute effective address formats, it is necessary to ensure that all memory operand references are resolved by the assembler (or by the linkage editor at the assembler's direction) into one of the program counter relative or address register indirect addressing modes. Avoiding ORG directives is not sufficient to ensure position independence, because it is possible for the assembler to produce absolute effective address formats even when no absolute symbols have been defined.

For example, if an instruction references a symbol that is not yet defined, or is defined either in another section or as an XREF in an unspecified section, the default action of the assembler is to direct the linkage editor to resolve the reference by supplying the absolute address of the symbol. By specifying OPT PCS, all references known to be in a relocatable section are resolved as a Program Counter (PC) relative address. However, this does not solve the problem of forward references, which would still default to absolute format. To override an absolute address mode when resolving the effective address format of an operand, the following formats may be used to force program counter relative addressing:

a. Forcing program counter with displacement

   An operand of the form: LABEL(PC)

   is resolved as a PC with displacement effective address, either by the assembler or by the linkage editor (at the assembler's direction). If LABEL cannot be resolved into a 16-bit displacement from the program counter, an error is generated.

b. Forcing PC with index plus displacement

   An operand of the form: LABEL(PC,Rn)

   is resolved as a PC with index plus displacement effective address by the assembler. Because the displacement in this mode is eight bits, the reference must be resolvable by the assembler. If LABEL cannot be resolved by the assembler into an 8-bit displacement from the program counter, an error is generated.

7.2 BASE-DISPLACEMENT ADDRESSING

Although PC relative addresses have the advantage of position-independence, such address formats often are not the most meaningful to the programmer when debugging an assembled module. There are many times when a programmer would prefer to see an address relative to a specified base -- i.e., in a base-displacement format. This is especially true when addressing tables, arrays,

and other data structures. Base-displacement references to a given location are "base relative" and, therefore, fixed with respect to a given base address; PC relative references to that same location are different in each instruction.

Base-displacement addressing must be handled explicitly by the programmer. For example, if the following data area is declared

```
TEMP      DS      $40
CONST     DC      $10
ARRAY1    DS.L    $10
ARRAY2    DS.L    $10
RESULT    DS.L    $10
```

the programmer may choose to load A6 with the address of TEMP and make references to the other data locations as displacements from this base address. For example, to move the first element of ARRAY1 to D1, the programmer may specify:

```
MOVE.L    ARRAY1-TEMP(A6),D1
```

Indexing with the low order contents of D0 may be added (as the array index):

```
MOVE.L    ARRAY1-TEMP(A6,D0),D1
```

7.3 BASE-DISPLACEMENT IN CONJUNCTION WITH FORCED POSITION INDEPENDENCE

Complete code-position independence can be achieved by using base-displacement addressing in conjunction with the PCS option and the forced PC relative addressing scheme outlined in paragraph 7-1. Although these techniques can be used to avoid all undesired absolute address formats, there are significant limitations of PC relative addressing in a position independent program, as noted below:

a. PC with displacement

   PC with displacement effective addresses are restricted only by the 16-bit displacement field. A displacement greater than 32K bytes from the current PC cannot be resolved in this format.

b. PC with index plus displacement

   The displacement field here is restricted to eight bits, limiting the range of this format to a 128-byte displacement from the current PC. This 8-bit displacement is not relocatable. Therefore, only symbols with a known displacement from the program counter may be resolved in a PC with index plus displacement format.

c. Operands in the alterable addressing category

   Neither PC relative mode is allowed as an alterable operand. This is a significant limitation in instructions which require an alterable operand, such as the destination operand in a MOVE instruction.

By appropriate use of base registers, these limitations can be overcome.

APPENDIX A

INSTRUCTION SET SUMMARY

This appendix provides a summary of the MC68000/MC68010/MC68020/MC68881 instruction set. For detailed information, refer to the M68000 16/32-bit Microprocessor Programmer's Reference Manual.

For the MC68881 only, the affected condition codes N Z I NAN are, respectively, bits 31, 30, 29, and 28 of the floating-point status register, rather than bits 4, 3, 2, 1, and 0 of the status MC68000/MC68010/MC68020 register. Thus, the four condition codes listed for MC68881 instructions refer to N Z I NAN, respectively.

Following are two instruction set summary tables -- one for the MC68000/MC68010/MC68020 and one for the MC68881.

INSTRUCTION SET SUMMARY - MC68000/MC68010/MC68020

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES X	N	Z	V	C
ABCD	Add decimal with extend	ABCD Dy,Dx ABCD -(Ay),-(Ax)	*	U	*	U	*
ADD	Add binary (NOTE 1)	ADD <ea>,Dn ADD Dn<ea>	*	*	*	*	*
ADDA	Add address	ADDA <ea>,An	-	-	-	-	-
ADDI	Add immediate	ADDI #<data>,<ea>	*	*	*	*	*
ADDQ	Add quick	ADDQ #<data>,<ea>	*	*	*	*	*
ADDX	Add extended	ADDX Dy,Dx ADDX -(Ay),-(Ax)	*	*	*	*	*
AND	AND logical	AND <ea>,Dn AND Dn,<ea>	-	*	*	0	0
ANDI	AND immediate	ANDI #<data>,<ea>	-	*	*	0	0
ASL, ASR	Arithmetic shift	ASd Dx,Dy ASd #<data>,Dy ASd <ea>	*	*	*	*	*
Bcc	Branch conditionally	Bcc <label>	-	-	-	-	-
BCHG	Test a bit and change	BCHG Dn,<ea> BCHG #<data>,<ea>	-	-	*	-	-
BCLR	Test a bit and clear	BCLR Dn,<ea> BCLR #<data>,<ea>	-	-	*	-	-
BFCHG	Complement bit field (MC68020)	BFCHG <ea>{<offset>:<width>}	-	*	*	0	0
BFCLR	Clear bit field (MC68020)	BFCLR <ea>{<offset>:<width>}	-	*	*	0	0

INSTRUCTION SET SUMMARY - MC68000/MC68010/MC68020 (cont'd)

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES X	N	Z	V	C
BFEXTS	Extract bit field signed (MC68020)	BFEXTS <ea>{<offset>:<width>},Dn	-	*	*	0	0
BFEXTU	Extract bit field unsigned (MC68020)	BFEXTU <ea>{<offset>:<width>},Dn	-	*	*	0	0
BFFFO	Find first one in bit field (MC68020)	BFFFO <ea>{<offset>:<width>},Dn	-	*	*	0	0
BFINS	Insert bit field (MC68020)	BFINS Dn,<ea>{<offset>:<width>}	-	*	*	0	0
BFSET	Set bit field (MC68020)	BFSET <ea>{<offset>:<width>},Dn	-	*	*	0	0
BFTST	Test bit field (MC68020)	BFTST <ea>{<offset>:<width>}	-	*	*	0	0
BKPT	Breakpoint (MC68020)	BKPT #<vector>	-	-	-	-	-
BRA	Branch always	BRA <label>	-	-	-	-	-
BSET	Test a bit and set	BSET Dn,<ea> BSET #<data>,<ea>	-	-	*	-	-
BSR	Branch to subroutine	BSR <label>	-	-	-	-	-
BTST	Test a bit	BTST Dn,<ea> BTST #<data>,<ea>	-	-	*	-	-
CALLM	Call module (MC68020)	CALLM #ddd,<ea>	-	-	-	-	-
CAS	Compare and swap with operand (MC68020)	CAS Dw,Do,<ea>	-	*	*	*	*
CAS2	Compare and swap with operand (MC68020)	CAS2 Dw1:Dw2,Do1:Do2,(Rz1):(Rz2)	-	*	*	*	*
CHK	Check register against bounds	CHK <ea>,Dn	-	*	U	U	U
CHK2	Check register against bounds (MC68020)	CHK2 <ea>,Rn	-	U	*	U	*

INSTRUCTION SET SUMMARY - MC68000/MC68010/MC68020 (cont'd)

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES X	N	Z	V	C
CLR	Clear an operand	CLR <ea>	-	0	1	0	0
CMP	Arithmetic compare	CMP <ea>,Dn	-	*	*	*	*
CMPA	Arithmetic compare address	CMPA <ea>,An	-	*	*	*	*
CMPI	Compare immediate	CMPI #<data>,<ea>	-	*	*	*	*
CMPM	Compare memory	CMPM (Ay)+,(Ax)+	-	*	*	*	*
CMP2	Compare register against bounds (MC68020)	CMP2 <ea>,Rn	-	U	*	U	*
DBcc	Test condition and decrement and branch (NOTE 2)	DBcc Dn,<label>	-	-	-	-	-
DIVS	Signed divide	DIVS <ea>,Dn	-	*	*	*	0
DIVU	Unsigned divide	DIVU <ea>,Dn	-	*	*	*	0
EOR	Exclusive OR logical	EOR Dn,<ea>	-	*	*	0	0
EORI	Exclusive OR immediate	EORI #<data>,<ea>	-	*	*	0	0
EXG	Exchange registers	EXG Rx,Ry	-	-	-	-	-
EXT	Sign extend	EXT Dn	-	*	*	0	0
EXTB	Sign extend byte (MC68020)	EXTB Dn	-	*	*	0	0
EXTW	Sign extend word (MC68020) (Part of EXT instruction)	EXTW Dn	-	*	*	0	0

INSTRUCTION SET SUMMARY - MC68000/MC68010/MC68020 (cont'd)

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES X	N	Z	V	C
JMP	Jump	JMP <ea>	-	-	-	-	-
JSR	Jump to subroutine	JSR <ea>	-	-	-	-	-
LEA	Load effective address	LEA <ea>,An	-	-	-	-	-
LINK	Link and allocate (NOTE 5)	LINK An,#<disp>	-	-	-	-	-
LSL, LSR	Logical shift	LSd Dx,Dy LSd #<data>,Dy LSd <ea>	*	*	*	0	*
MOVE	Move data from source to destination	MOVE <ea>,<ea>	-	*	*	0	0
MOVE to SR	Move to the status register	MOVE <ea>,SR	*	*	*	*	*
MOVE from SR	Move from the status register	MOVE SR,<ea>	-	-	-	-	-
MOVE to CC	Move to condition codes	MOVE <ea>,CCR	*	*	*	*	*
MOVE from CC	Move from condition codes (MC68010 or newer)	MOVE CCR,<ea>	-	-	-	-	-
MOVE USP	Move user stack pointer	MOVE USP,An MOVE An,USP	-	-	-	-	-
MOVEA	Move address	MOVEA <ea>,An	-	-	-	-	-
MOVEC	Move to/from control register (MC68010 or newer) (NOTE 3)	MOVEC Rc,Rn MOVEC Rn,Rc	-	-	-	-	-

INSTRUCTION SET SUMMARY - MC68000/MC68010/MC68020 (cont'd)

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES X	N	Z	V	C
MOVEM	Move multiple registers (NOTE 4)	MOVEM <register list>,<ea> MOVEM <ea>,<register list>	-	-	-	-	-
MOVEP	Move peripheral data	MOVEP Dx,d(Ay) MOVEP d(Ay),Dx	-	-	-	-	-
MOVEQ	Move quick	MOVEQ #<data>,Dn	-	*	*	0	0
MOVES	Move to/from address (MC68010 or newer)	MOVES <ea>,Rn MOVES Rn,<ea>	-	-	-	-	-
MULS	Signed multiply	MULS <ea>,Dn	-	*	*	0	0
MULU	Unsigned multiply	MULU <ea>,Dn	-	*	*	0	0
NBCD	Negate decimal with extend	NBCD <ea>	*	U	*	U	*
NEG	2's complement negation	NEG <ea>	*	*	*	*	*
NEGX	Negate with extend	NEGX <ea>	*	*	*	*	*
NOP	No operation	NOP	-	-	-	-	-
NOT	Logical complement	NOT <ea>	-	*	*	0	0
OR	Inclusive OR logical	OR <ea>,Dn OR Dn,<ea>	-	*	*	0	0
ORI	Inclusive OR immediate	ORI #<data>,<ea>	-	*	*	0	0
PACK	Pack BCD (MC68020)	PACK -(Ay),-(Ax) PACK Dy,Dx	-	-	-	-	-
PEA	Push effective address	PEA <ea>	-	-	-	-	-

INSTRUCTION SET SUMMARY - MC68000/MC68010/MC68020 (cont'd)

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES X	N	Z	V	C
RESET	Reset external devices	RESET	-	-	-	-	-
ROL, ROR	Rotate without extend	ROd Dx,Dy ROd #<data>,Dy ROd <ea>	-	*	*	0	*
ROXL, ROXR	Rotate with extend	ROXd Dx,Dy ROXd #<data>,Dy ROXd <ea>	*	*	*	0	*
RTD	Return from subroutine with displacement (MC68010 or newer) (NOTE 5)	RTD #<disp>	-	-	-	-	-
RTE	Return from exception	RTE	*	*	*	*	*
RTM	Return from module (MC68020)	RTM Rn	-	-	-	-	-
RTR	Return and restore condition codes	RTR	*	*	*	*	*
RTS	Return from subroutine	RTS	-	-	-	-	-
SCD	Subtract decimal with extend	SBCD Dy,Dx SBCD -(Ay),-(Ax)	*	U	*	U	*
Scc	Set according to condition	Scc <ea>	-	-	-	-	-
STOP	Stop program execution	STOP #<data>	-	-	-	-	-
SUB	Subtract binary	SUB <ea>,Dn SUB Dn,<ea>	*	*	*	*	*
SUBA	Subtract address	SUBA <ea>,An	-	-	-	-	-

INSTRUCTION SET SUMMARY - MC68000/MC68010/MC68020 (cont'd)

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES X	N	Z	V	C
SUBI	Subtract immediate	SUBI #<data>,<ea>	*	*	*	*	*
SUBQ	Subtract quick	SUBQ #<data>,<ea>	*	*	*	*	*
SUBX	Subtract with extend	SUBX Dy,Dx SUBX -(Ay),-(Ax)	*	*	*	*	*
SWAP	Swap register halves	SWAP Dn	-	*	*	0	0
TAS	Test and set an operand	TAS <ea>	-	*	*	0	0
Tcc	Trap on condition code (MC68020)	Tcc	-	-	-	-	-
TDIVS	Truncated signed (MC68020)	TDIVS <ea>,{Di: }Dj	-	*	*	*	0
TDIVU	Truncated unsigned divide (MC68020)	TDIVU <ea>,{Di: }Dj	-	*	*	*	0
TPcc	Trap on condition code (MC68020)	TPCC #xxx	-	-	-	-	-
TRAP	Trap	TRAP #<vector>	-	-	-	-	-
TRAPV	Trap on overflow	TRAPV	-	-	-	-	-
TST	Test an operand	TST <ea>	-	*	*	0	0

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES X	N	Z	V	C
UNLK	Unlink	UNLK An	-	-	-	-	-
UNPK	Unpack BCD (MC68020)	UNPK -(Ay),-(Ax)	-	-	-	-	-
		UNPK Dy,Dx	-	-	-	-	-

NOTES:

1. <ea> specifies effective address.
2. The assembler accepts DBRA for the F (never true) condition.
3. Rc specifies control register.
4. <register list> specifies the registers selected for transfer to or from memory. <register list> may be:

   Rn - a single register.

   Rn-Rm - a range of consecutive registers with m being greater than n.

   Any combination of the above, separated by a slash.

5. <disp> is a 2's complement integer, 16 bits in size, which is sign extended to 32 bits before adding to the stack pointer.

INSTRUCTION SET SUMMARY - MC68881

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES N	Z	I	NAN
FABS	Absolute value function	FABS <ea>,FPn FABS FPm,FPn FABS FPn	*	*	*	*
FACOS	Arccosine function	FACOS <ea>,FPn FACOS FPM,FPn FACOS FPn	*	*	*	*
FADD	Floating point add	FADD <ea>,FPn FADD FPm,FPn	*	*	*	*
FASIN	Arcsine function	FASIN <ea>,FPn FASIN FPm,FPn FASIN FPn	*	*	*	*
FATAN	Arctangent function	FATAN <ea>,FPn FATAN FPm,FPn	*	*	*	*
FATANH	Hyperbolic arctangent function	FATANH <ea>,FPn FATANH FPm,FPn FATANH FPn	*	*	*	*
FBfpcc	Co-processor branch conditionally (MC68881)	FBfpcc <label>	-	-	-	- -
FCMP	Floating point compare	FCMP <ea>,FPn FCMP FPm,FPn	*	*	*	*
FCOS	Cosine function	FCOS <ea>,FPn FCOS FPm,FPn FCOS FPn	*	*	*	*
FCOSH	Hyperbolic cosine function	FCOSH <ea>,FPN FCOSH FPm,FPn FCOSH FPn	*	*	*	*

INSTRUCTION SET SUMMARY - MC68881 (cont'd)

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES N	Z	I	NAN
FDBfpcc	Decrement and branch on condition (MC68881)	FDBfpcc DN,<label>	-	-	-	-
FDIV	Floating point divide	FDIV <ea>,FPn FDIV FPm,FPn	*	*	*	*
FETOX	e**x function	FETOX <ea>,FPn FETOX FPm,FPn FETOX FPn	*	*	*	*
FETOXM1	e**x(x-1) function	FETOXM1 <ea>,FPn FETOXM1 FPm,FPn FETOXM1 FPn	*	*	*	*
FGETEXP	Get the exponent function	FGETEXP <ea>,FPn FGETEXP FPm,FPn FGETEXP FPn	*	*	*	*
FGETMAN	Get the Mantissa function	FGETMAN <ea>,FPn FGETMAN FPm,FPn FGETMAN FPn	*	*	*	*
FINT	Integer part function	FINT <ea>,FPn FINT FPm,FPn FINT FPn	*	*	*	*
FLOG2	Binary log function	FLOG2 <ea>,FPn FLOG2 FPm,FPn FLOG2 FPn	*	*	*	*
FLOG10	Common log function	FLOG10 <ea>,FPn FLOG10 FPm,FPn FLOG10 FPn	*	*	*	*

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES N	Z	I	NAN
FLOGN	Natural log function	FLOGN <ea>,FPn FLOGN FPm,FPn FLOGN FPn	*	*	*	*
FLOGNP1	Natural log (x+1) function	FLOGNP1 <ea>,FPn FLOGNP1 FPm,FPn FLOGNP1 FPn	*	*	*	*
FMOD	Floating point module	FMOD <ea>,FPn FMOD FPm,FPn	*	*	*	*
FMOVE	Move to floating point register from memory or another floating floating point register	FMOVE <ea>,FPn FMOVE FPm,FPn	*	*	*	*
	Move from floating point register to memory	FMOVE FPN,<ea> FMOVE.P FPn,<ea>{#k} FMOVE.P FPn,<ea>{Dn}				
	Move to/from memory from/to special register	FMOVE <ea>,CONTROL|STATUS|IADDR				
		FMOVE CONTROL|STATUS|IADDR,<ea>				
FMOVECR	Move a ROM-stored to a floating point register	FMOVECR #ccc,FPn		*	*	* *

INSTRUCTION SET SUMMARY - MC68881 (cont'd)

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES N	Z	I	NAN
FMOVEM	Move to multiple floating point registers	FMOVEM <ea>,<fp_reg_list>	*	*	*	*
	Move to a data register or special register	FMOVEM <ea>,Dn FMOVEM <ea>,CONTROL/STATUS/IADDR				
	Move from multiple floating point registers to memory	FMOVEM <fp_reg_list>,<ea>				
	Move from data register or special register to memory	FMOVEM Dn,<ea> FMOVEM CONTROL/STATUS/IADDR,<ea>				
FMUL	Floating point multiply	FMUL <ea>,FPn FMUL FPm,FPn	*	*	*	*
FNEG	Negate function	FNEG <ea>,FPn FNEG FPm,FPn FNEG FPn	*	*	*	*
FNOP	Floating point NO-OP	FNOP	*	*	*	*
FREM	Floating point remainder	FREM <ea>,FPn FREM FPM,FPn	*	*	*	*
FRESTORE	Restore internal state of co-processor (MC68881)	FRESTORE <ea>	-	-	-	-
FSAVE	Co-processor save (MC68881)	FSAVE <ea>	-	-	-	-
FSCALE	Floating point scale exponent	FSCALE <ea>,FPn FSCALE FPm,FPn	*	*	*	*
FSfpcc	Set on condition (MC68881)	FSfpcc <ea>	-	-	-	-

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES N	Z	I	NAN
FTfpcc	Trap on condition without a parameter (MC68881)	FTfpcc	-	-	-	-
FTPfpcc	Trap on condition with a parameter (MC68881)	FTPfpcc #xxx	-	-	-	-
FSGLDIV	Floating point single precision divide	FSGLDIV <ea>,FPn FSGLDIV FPm,FPn	*	*	*	*
FSGLMUL	Floating point single precision multiply	FSGLMUL <ea>,FPn FSGLMUL FPm,FPn	*	*	*	*
FSIN	Sine function	FSIN <ea>,FPn FSIN FPm,FPn FSIN FPn	*	*	*	*
FSINCOS	Sine/cosine function	FSINCOS <ea>,FPm:FPn	*	*	*	*
FSINH	Hyperbolic sine function	FSINH <ea>,FPn FSINH FPm,FPn FSINH FPn	*	*	*	*
FSQRT	Square root function	FSQRT <ea>,FPn FSQRT FPm,FPn FSQRT FPn	*	*	*	*
FSUB	Floating point subtract	FSUB <ea>,FPn FSUB FPm,FPn	*	*	*	*
FTAN	Tangent function	FTAN <ea>,FPn FTAN FPm,FPn FTAN FPn	*	*	*	*

INSTRUCTION SET SUMMARY - MC68881 (cont'd)

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	CONDITION CODES N	Z	I	NAN
FTANH	Hyperbolic tangent function	FTANH <ea>,FPn FTANH FPm,FPn FTANH FPn	*	*	*	*
FTENTOX	10**x function	FTENTOX <ea>,FPn FTENTOX FPm,FPn FTENTOX FPn	*	*	*	*
FTEST	Floating point test an operand	FTEST <ea>	*	*	*	*
FTWOTOX	2**x function	FTWOTOX <ea>,FPn FTWOTOX FPm,FPn FTWOTOX FPn	*	*	*	*
FYTOX	Floating point y**x	FYTOX <ea>,FPn FYTOX FPm,FPn	*	*	*	*

APPENDIX B

CHARACTER SET

The character set recognized by the Motorola M68000 Family Resident Structured Assembler is a subset of ASCII (American Standard Code for Information Interchange, 1968). The characters listed below are recognized by the assembler, and the ASCII code is shown on the following pages.

1. The uppercase letters A through Z

2. The lowercase letters a through z (MC68020 assemblers or SYSTEM V/68 only)

3. The integers 0 through 9

4. Four arithmetic operators: + - * /

5. The logical operators: >> << & !

6. Parentheses used in expressions ( )

7. Characters used as special prefixes:

   # (pound sign) specifies the immediate mode of addressing
   $ (dollar sign) specifies a hexadecimal number
   @ (commercial "at") specifies an octal number
   % (percent) specifies a binary number
   ' (apostrophe) specifies an ASCII literal character

8. The special characters used in macros: < > \ @

9. Four separating characters:

   (space)
   (tab) (M68020 assemblers or SYSTEM V/68 only)
   , (comma)
   . (period)

10. A comment in a source statement may include any characters with ASCII hexadecimal values from 20 (SP) through 7E (~).

11. Character used as a special suffix:

    : (colon) specifies the end of a label

ASCII Character Set

CHARACTER	COMMENTS	HEX VALUE
NUL	Null or tape feed	00
SOH	Start of Heading	01
STX	Start of Text	02
ETX	End of Text	03
EOT	End of Transmission	04
ENQ	Enquire (who are you, WRU)	05
ACK	Acknowledge	06
BEL	Bell	07
BS	Backspace	08
HT	Horizontal Tab	09
LF	Line Feed	0A
VT	Vertical Tab	0B
FF	Form Feed	0C
RETURN	Carriage Return	0D
SO	Shift Out (to red ribbon)	0E
SI	Shift In (to black ribbon)	OF
DLE	Data Link Escape	10
DC1	Device Control 1	11
DC2	Device Control 2	12
DC3	Device Control 3	13
DC4	Device Control 4	14
NAK	Negative Acknowledge	15
SYN	Synchronous idle	16
ETB	End of Transmission Block	17
CAN	Cancel	18
EM	End of Medium	19
SUB	Substitute	1A
ESC	Escape, prefix	1B
FS	File Separator	1C
GS	Group Separator	1D
RS	Record Separator	1E
US	Unit Separator	1F

ASCII Character Set (cont'd)

CHARACTER	COMMENTS	HEX VALUE
SP	Space or blank	20
!	Exclamation point	21
"	Quotation marks (dieresis)	22
#	Number sign	23
$	Dollar sign	24
%	Percent sign	25
&	Ampersand	26
'	Apostrophe (acute accent, closing single quote)	27
(	Opening parenthesis	28
)	Closing parenthesis	29
*	Asterisk	2A
+	Plus sign	2B
,	Comma (cedilla)	2C
-	Hyphen (minus)	2D
.	Period (decimal point)	2E
/	Slant	2F
0	Digit 0	30
1	Digit 1	31
2	Digit 2	32
3	Digit 3	33
4	Digit 4	34
5	Digit 5	35
6	Digit 6	36
7	Digit 7	37
8	Digit 8	38
9	Digit 9	39
:	Colon	3A
;	Semicolon	3B
<	Less than	3C
=	Equals	3D
>	Greater than	3E
?	Question mark	3F

ASCII Character Set (cont'd)

CHARACTER	COMMENTS	HEX VALUE
@	Commercial at	40
A	Uppercase letter A	41
B	Uppercase letter B	42
C	Uppercase letter C	43
D	Uppercase letter D	44
E	Uppercase letter E	45
F	Uppercase letter F	46
G	Uppercase letter G	47
H	Uppercase letter H	48
I	Uppercase letter I	49
J	Uppercase letter J	4A
K	Uppercase letter K	4B
L	Uppercase letter L	4C
M	Uppercase letter M	4D
N	Uppercase letter N	4E
O	Uppercase letter O	4F
P	Uppercase letter P	50
Q	Uppercase letter Q	51
R	Uppercase letter R	52
S	Uppercase letter S	53
T	Uppercase letter T	54
U	Uppercase letter U	55
V	Uppercase letter V	56
W	Uppercase letter W	57
X	Uppercase letter X	58
Y	Uppercase letter Y	59
Z	Uppercase letter Z	5A
[	Opening bracket	5B
\	Reverse slant	5C
]	Closing bracket	5D
^	Circumflex	5E
_	Underline	5F

ASCII Character Set (cont'd)

CHARACTER	COMMENTS	HEX VALUE
'	Quotation mark	60
a	Lowercase letter a	61
b	Lowercase letter b	62
c	Lowercase letter c	63
d	Lowercase letter d	64
e	Lowercase letter e	65
f	Lowercase letter f	66
g	Lowercase letter g	67
h	Lowercase letter h	68
i	Lowercase letter i	69
j	Lowercase letter j	6A
k	Lowercase letter k	6B
l	Lowercase letter l	6C
m	Lowercase letter m	6D
n	Lowercase letter n	6E
o	Lowercase letter o	6F
p	Lowercase letter p	70
q	Lowercase letter q	71
r	Lowercase letter r	72
s	Lowercase letter s	73
t	Lowercase letter t	74
u	Lowercase letter u	75
v	Lowercase letter v	76
w	Lowercase letter w	77
x	Lowercase letter x	78
y	Lowercase letter y	79
z	Lowercase letter z	7A
{	Opening brace	7B
|	Vertical line	7C
}	Closing brace	7D
~	Equivalent	7E
DEL	Delete	7F

APPENDIX C

SAMPLE ASSEMBLER OUTPUT

```
MOTOROLA M68000 ASM      FIX : 108.DEMO    .MAIN    .SA


  1                          *
  2                          MAIN     IDNT     2,3                    Demonstration Program
  3                          *
  4                          *        This program counts occurrences of vowels (A,E,I,O,U)
  5                          *        in the command line and outputs an error if fewer than 10
  6                          *        vowels are found in the command line, aside from the vowels
  7                          *        in the program name 'TSTPROG'.
  8                          *        It is written in a contrived fashion to illustrate several
  9                          *        features of the M68000 assembler.
 10                          *
 11                                   OPT      CRE                    ! Create a cross-reference listing
 12                                   OPT      MEX                    ! Enable macro expansions
 13                          *
 14                                   XREF.S   15:VOWEL               ! Array containing vowel count info
 15                                   XREF.S   15:STACK               ! Scratch stack space
 16                                   XREF     FINDV                  ! Routine that does the counting
 17                                   XREF     CMDLEN                 ! Length of the command line
 18                          *
 19                          *        These are offsets into the vowel array contained in module FINDV.
 20                          *        Each entry in this array contains 1 byte for the vowel's name
 21                          *        and one byte for the count of occurrences of the vowel.
 22                          *
 23                                   OFFSET   0
 24   00000000 00000002      A        DS.W     1
 25   00000002 00000002      E        DS.W     1
 26   00000004 00000002      I        DS.W     1
 27   00000006 00000002      O        DS.W     1
 28   00000008 00000002      U        DS.W     1
 29                          *
 30                          *        This macro calls FINDV to count occurrences of the vowel
 31                          *        contained in argument 1.  It then adds that subtotal into
 32                          *        the running total contained in D1.
 33                          *
 34                          CHKVOWEL MACRO
 35                                   MOVE.B   #\1,D0          ! Store current vowel offset into VOWEL
 36                                   JSR      FINDV           ! Find all occurrences of it
 37                                   ADD.B    \1+1(A0),D1     ! Add this to the total vowel count
 38                                   ENDM
 39                          *
 40
 41             00000008              SECTION  8
 42 8           00000000    START     EQU      *
 43
 44 8 00000000 5346                   SUB.W    #1,D6                  ! Index command line from offset 0 and not 1
 45 8 00000002 33C600000000           MOVE.W   D6,CMDLEN              ! Save the command line len
 46                          *                                        ! as passed by VERSAdos
 47 8 00000008 4FF80000               LEA      STACK,A7               ! Initialize the stack area
 48 8 0000000C 41F80000               LEA      VOWEL,A0               ! Start of the  vowel table
 49 8 00000010 4241                   CLR.W    D1                     ! Current total vowel count
 50 8 00000012 4280                   CLR.L    D0                     ! Will hold offset to current char later
 51
 52 8 00000014                        CHKVOWEL A
    8 00000014 103C0000               MOVE.B   #A,D0                  ! Store current vowel offset into VOWEL
    8 00000018 4EB900000000           JSR      FINDV                  ! Find all occurrences of it
    8 0000001E D2280001               ADD.B    A+1(A0),D1             ! Add this to the total vowel count
 53
 54 8 00000022                        CHKVOWEL E
    8 00000022 103C0002               MOVE.B   #E,D0                  ! Store current vowel offset into VOWEL
    8 00000026 4EB900000000           JSR      FINDV                  ! Find all occurrences of it
    8 0000002C D2280003               ADD.B    E+1(A0),D1             ! Add this to the total vowel count
 55
```

```
  56 8 00000030                          CHKVOWEL I
     8 00000030 103C0004                 MOVE.B   #I,D0              ! Store current vowel offset into VOWEL
     8 00000034 4EB900000000             JSR      FINDV              ! Find all occurrences of it
     8 0000003A D2280005                 ADD.B    I+1(A0),D1         ! Add this to the total vowel count
  57
  58 8 0000003E                          CHKVOWEL O
     8 0000003E 103C0006                 MOVE.B   #O,D0              ! Store current vowel offset into VOWEL
     8 00000042 4EB900000000             JSR      FINDV              ! Find all occurrences of it
     8 00000048 D2280007                 ADD.B    O+1(A0),D1         ! Add this to the total vowel count
  59
  60 8 0000004C                          CHKVOWEL U
     8 0000004C 103C0008                 MOVE.B   #U,D0              ! Store current vowel offset into VOWEL
     8 00000050 4EB900000000             JSR      FINDV              ! Find all occurrences of it
     8 00000056 D2280009                 ADD.B    U+1(A0),D1         ! Add this to the total vowel count
  61
  62                                     IF.B     #10 <GT> D1 THEN.S ! Not enough vowels
  63 8 00000060 3041                       MOVE.W   D1,A0
  64 8 00000062 700E                       MOVE.L   #14,D0
  65 8 00000064 4E41                       TRAP     #1                ! generate error showing # of vowels found
  66 8 00000066 0000                       DC.W     0
  67                                     ENDI
  68
  69 8 00000068 700F                     MOVE.L   #15,D0
  70 8 0000006A 4E41                     TRAP     #1                ! Exit gracefully if all is OK
  71
  72 8          00000000                 END      START

****** TOTAL ERRORS      0--
****** TOTAL WARNINGS    0--
```

SYMBOL TABLE LISTING

SYMBOL NAME		SECT	VALUE	CROSS-REF (LINENUMBERS)						
A			00000000	-24	52					
CHKVOWEL	MACR	*		-34	11	52	54	56	58	60
CMDLEN	XREF	*	00000000	-17	45					
E			00000002	-25	54					
FINDV	XREF	*	00000000	-16	52	54	56	58	60	
I			00000004	-26	56					
O			00000006	-27	58					
STACK	XREF	F	00000000	-15	47					
START		8	00000000	-42	72					
U			00000008	-28	60					
VOWEL	XREF	F	00000000	-14	48					
Z_L1.000		8	00000068	-67	62					

```
  1                         *
  2                         FINDV   IDNT    1,1                     Routine subordinate to MAIN
  3                         *
  4                         *       This routine counts occurrences of a given vowel.  The vowel
  5                         *       is identified by an offset into the vowel table.  This offset
  6                         *       is stored in D0.
  7                         *       This routine is written in a contrived fashion to illustrate several
  8                         *       features of the M68000 assembler.
  9                         *
 10                                 OPT     CRE                     ! Create a cross-reference listing
 11                                 OPT     CEX                     ! Print DC expansions
 12                        *
 13                                 XDEF    VOWEL,FINDV,STACK,CMDLEN,CMDSTR
 14                        *
 15              0000000F           SECTION.S 15
 16                        *
 17                        *       Register save area
 18                        *
 19 F 00000000 00000040    RSAVE   DS.L    8*2
 20                        *
 21                        *       Stack area for the program
 22 F 00000040 00000050             DS.L    20
 23 F 00000090 00000004    STACK   DS.L    1
 24                        *
 25                        *       Following is the vowel array VOWEL.
 26                        *       Each entry in this array contains 1 byte for the vowel's name
 27                        *       and one byte for the count of occurrences of the vowel.
 28                        *
 29 F 00000094 4100        VOWEL   DC.B    'A',0
 30 F 00000096 4500                DC.B    'E',0
 31 F 00000098 4900                DC.B    'I',0
 32 F 0000009A 4F00                DC.B    'O',0
 33 F 0000009C 5500                DC.B    'U',0
 34                        *
 35                        *       Next is the area which holds the command line length and string.
 36                        *
 37 F 0000009E 0000        CMDLEN  DC.W    0
 38 F 000000A0 000000A0    CMDSTR  COMLINE 160
 39                        *
 40
 41              00000008          SECTION 8
 42                        *
 43                        *       On entry to this routine, D0 contains the offset to the start
 44                        *       of the current entry in the vowel table.
 45                        *       This routine then tallies occurrences of the given vowel and
 46                        *       stores that value in the table.
 47                        *
 48                        SAVEREG REG     D0-D3/A0-A2
 49                        *
 50 8            00000000 FINDV   EQU     *
 51 8 00000000 48F8070F0000         MOVEM.L SAVEREG,RSAVE           ! Save all registers we are using
 52
 53 8 00000006 41F80094             LEA     VOWEL,A0
 54 8 0000000A 12300000             MOVE.B  0(A0,D0.W),D1           ! Value of this vowel
 55 8 0000000E 41F00001             LEA     1(A0,D0.W),A0           ! Addr of counter for this vowel
 56 8 00000012 43F800A0             LEA     CMDSTR,A1               ! Addr of command line string
 57
 58                                 FOR     D3 = #0 TO CMDLEN BY #1 DO
    8 0000001A 6000000E               BRA.    Z_L2.000
****** WARNING 550--
 59 8 0000001E 14313000             MOVE.B  (A1,D3.W),D2            ! Current char is now in D2
 60
 61                                 IF.B    D1 <EQ> D2 THEN.S
 62 8 00000026 5210                   ADDQ.B  #1,(A0)               ! Tally matching chars
 63                                 ENDI
```

```
  64
  65                                    ENDF
  66
  67 8 00000030 4CF8070F0000            MOVEM.L  RSAVE,SAVEREG        ! Restore registers we used
  68 8 00000036 4E75                    RTS
  69
  70                                    END

****** TOTAL ERRORS      0--  58
****** TOTAL WARNINGS    1--  58


SYMBOL TABLE LISTING

SYMBOL NAME      SECT   VALUE       CROSS-REF (LINENUMBERS)


CMDLEN      XDEF   F   0000009E    -37    -13     65
CMDSTR      XDEF   F   000000A0    -38    -13     56
FINDV       XDEF   8   00000000    -50    -13
RSAVE              F   00000000    -19     51     67
SAVEREG     REG    *               -48     51     67
STACK       XDEF   F   00000090    -23    -13
VOWEL       XDEF   F   00000094    -29    -13     53
Z_L1.001           8   0000001E    -58     65
Z_L1.002           8   00000028    -63     61
Z_L2.000           8   0000002A    -65     58
```

APPENDIX D

EXAMPLE OF LINKED ASSEMBLY-LANGUAGE PROGRAMS UNDER VERSAdos

```
Motorola M68000 Linkage Editor

Command Line:

LINK 108.DEMO.MAIN/108.DEMO.FINDV,TSTPROG,TSTPROG;HIMUX

Options in Effect:  -A,-B,-D,H,I,-L,M,O,P,-Q,-R,-S,-U,-W,X

User Commands: None

Object Module Header Information:

Module     Ver Rev Language Date     Time     Creation File Name

MAIN         2   3 Assembly 09/13/82 13:12:27 FIX:108.DEMO.MAIN.SA
           Demonstration Program

FINDV        1   1 Assembly 09/13/82 13:12:54 FIX:108.DEMO.FINDV.SA
           Routine subordinate to MAIN

Load Map:


Segment SEG1(R): 00000000 000000FF 8,9,10,11,12,13,14
Module      S   T  Start     End       Externally Defined Symbols

MAIN         8     00000000  0000006B
FINDV        8     0000006C  000000A3  FINDV      0000006C


Segment SEG2: 00000100 000002FF 15
Module      S   T  Start     End       Externally Defined Symbols

FINDV       15  S  00000100  0000023F  CMDLEN     0000019E  CMDSTR     000001A0
                                       VOWEL      00000194  STACK      00000190


Table of Externally Defined Symbols:

Name        Address   Module      Displ     Sect Seg   Library      Input

CMDLEN      0000019E  FINDV       0000009E  15   SEG2               FINDV   .RO
CMDSTR      000001A0  FINDV       000000A0  15   SEG2               FINDV   .RO
FINDV       0000006C  FINDV       00000000   8   SEG1               FINDV   .RO
STACK       00000190  FINDV       00000090  15   SEG2               FINDV   .RO
VOWEL       00000194  FINDV       00000094  15   SEG2               FINDV   .RO


Unresolved References: None


Multiply Defined Symbols: None


Lengths (in bytes):

     Segment      Hex          Decimal

      SEG1     00000100           256
      SEG2     00000200           512
Total Length   00000300           768
```

```
No Errors
No Warnings

Load module has been created.
```

APPENDIX E

ASSEMBLY ERROR CODES

Error messages generated during an assembly may originate from the assembler or from Pascal or the operating system environment. Assembler-generated messages may be of two forms:

1. ****** ERROR xxx -- nnnn

   where xxx is the number of the error (defined in the list in this appendix), and nnnn is the number of the line where the previous error occurred.

   Errors indicate that the assembler is unable to interpret or implement the intent of a source line.

2. ****** WARNING xxx -- nnnn

   where xxx is the number of the error (defined in the list in this appendix), and nnnn is the number of the line where the previous error occurred.

   Warnings may indicate possible recoverable errors in the source code, or that a more optimal instruction format is possible.

ERROR CODE	MEANING OF ERROR
	SYNTACTIC ERRORS
200	Illegal character (in context)
201	Size code/extension is invalid
202	Syntax error
203	Size code/extension not allowed
204	Label required
205	End directive missing
206	Register ranges must be specified in increasing order (e.g., A1-A3, D0-D7, FP2-FP6)
207	A and D registers can't be intermixed in a MOVEM register range
208	In the register pair Di:Dj, Di must be distinct from Dj.

ERROR CODE	MEANING OF ERROR
	OPERAND/ADDRESS MODE ERRORS
210	Missing operand(s)
211	Too many operands for this instruction
212	Improper termination of operand field
213	Illegal address mode for this operand
214	Illegal forward reference
215	Symbol/expression must be absolute
216	Immediate source operand required
217	Illegal register for this instruction
218	Illegal operation on a relative symbol
219	Memory shifts may be only single bit
220	Invalid shift count
221	Invalid section number
222	"{o:w}" or "{k}" expression not allowed here
223	Too many registers found in an M68020 addressing mode form
224	Too many expressions found in an M68020 addressing mode form
225	More than one pair of [ ]s found in an M68020 addressing mode form
226	"{o:w}" expression expected in this instruction
	SYMBOL DEFINITION
230	Attempt to redefine a reserved symbol
231	Attempt to redefine a macro; new definition ignored
232	Attempt to redefine the command line location
233	Command line length must be > 0; ignored
234	Redefined symbol
235	Undefined symbol
236	Phasing error on PASS2
237	Start address must be in this module, if specified
238	Undefined operation (opcode)
239	Named common symbol may not be XDEF

ERROR CODE	MEANING OF ERROR
	DATA SIZE RESTRICTIONS
250	Displacement size error
251	Value too large
252	Address too large for forced absolute short
253	Byte mode not allowed for this opcode
254	Multiplication overflow
255	Division by zero
256	Value out of range
257	Branch to odd address detected
	MACRO ERRORS
260	Misplaced MACRO, MEXIT, or ENDM directive
261	Macro definitions may not be nested
262	Illegal parameter designation
263	A period may occur only as the first character in a macro name
264	Missing parameter reference
265	Too many parameters in this macro call
266	Reference precedes macro definition
267	Overflow of input buffer during macro text expansion
	CONDITIONAL ASSEMBLY ERRORS
270	Unexpected 'ENDC'
271	Bad ending to conditional assembly structure (ENDC expected)

ERROR CODE	MEANING OF ERROR
	STRUCTURED SYNTAX ERRORS
280	Misplaced structured control directive (ignored)
281	Missing "ENDI"
282	Missing "ENDF"
283	Missing "ENDW"
284	Missing "UNTIL"
285	Unresolved syntax error in the preceding parameterized structured control directive; recovery attempted with the current line
286	"=" Expected; characters up to "=" ignored
287	"<" Expected; characters up to "<" ignored
288	">" Expected; characters up to ">" ignored
289	"DO" expected; remainder of line ignored
290	"THEN" expected; remainder of line ignored
291	"TO" or "DOWNTO" expected; "TO" assumed
292	Illegal condition code specified
	MISCELLANEOUS
300	Implementation restriction
301	Too many relocatable symbols referenced <linkage editor restricted>
302	Relocation of byte field attempted
303	Absolute section of length zero defined (link error)
304	Nested "INCLUDE" files not allowed; ignored
305	File name required in operand field
310	Illegal syntax for "P=nnnnn' option - option ignored
311	Illegal processor number for 'P=nnnnn' option - option ignored
312	Processor option does not agree with command line option -- option ignored
313	This directive is not valid for the processor that is currently specified.
314	An "OFFSET" block must be followed by an "ORG" or "SECTION" before more code is generated.

ERROR CODE	MEANING OF ERROR
	FLOATING POINT ERRORS
330	Type (size) incompatibility exists between an operand and the opcode size.
331	Exponent string is too long. Will be truncated on the right which will almost certainly return the wrong value.
332	A non-decimal character was found in the decimal string. The haracter will be ignored and the conversion will continue although the results should be highly suspect.
333	The input decimal string is too big to be represented in the specified size. Infinity or the largest positive or negative number will be returned depending on the sign and current rounding mode.
334	The input decimal string is too small to be represented in the specified size. It was denormalized or reduced to zero.
	INTERNAL ERRORS
400 • • • 499	
	SOURCE CODE NOT OPTIMAL OR RECOVERABLE ERRORS
500	This byte will be sign-extended to 32 bits
501	Missing parameter reference in macro source
502	Too many parameters in this macro call
503	Warning - processor type should not be changed after any executable code is generated
504	Warning - processor type should not be changed after the user once sets it
550	This branch can also allow a word extension
551	This absolute address could be short
552	This expression/displacement could be represented in 16 bits rather than 32 bits.
553	Warning - this instruction may cause a branch to an odd address

ERROR CODE	MEANING OF ERROR
	FLOATING POINT WARNINGS
700	Mantissa string is too long. It will be truncated after 17 digits.
701	Decimal strings can be guaranteed to be accurate only to double precision in the worst case. In the best case, they are accurate to extended precision.
702	The decimal string to fp conversion was inexact (some rounding error occurred). This may or may not be important to the user.
703	Use of the L, D, X, and P extensions in the FSGLDIV and FSGLMUL instructions may result in a loss of accuracy.

NOTE

If more than 10 errors occur in one line, the message

***** too many errors on this line

will be generated.

SUGGESTION/PROBLEM REPORT

MICROSYSTEMS

QUALITY • PEOPLE • PERFORMANCE

Motorola welcomes your comments on its products and publications. Please use this form.

To: Motorola Inc.
Microsystems
2900 S. Diablo Way
Tempe, Arizona 85282
Attention: Publications Manager
Maildrop DW164

Product: _____ Manual: _____

COMMENTS: _____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please Print

Name _____ Title _____

Company _____ Division _____

Street _____ Mail Drop _____ Phone _____

City _____ State _____ Zip _____

For Additional Motorola Publications
Literature Distribution Center
616 West 24th Street
Tempe, AZ 85282
(602) 994-6561

Four Phase/Motorola Customer Support, Tempe Operations
(800) 528-1908
(602) 438-3100

MOTOROLA

16867 PRINTED IN USA (6/84) MESSENGER 5M