



# Macro/1000

## Reference Manual

**RTE-6/VM • RTE-XL • RTE-A  
HP 1000 Computer Systems**



# PRINTING HISTORY

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, Update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past Updates, however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all Updates.

To determine what manual edition and update is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog, Software Product Catalog, or Diagnostic Configurator Manual.

First Edition ..... Oct 1981  
Update 1 ..... Apr 1982  
Update 2 ..... Jul 1982

## NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

**HP Computer Museum**  
**[www.hpmuseum.net](http://www.hpmuseum.net)**

**For research and education purposes only.**

# Preface

This manual describes the Macro/1000 Assembly Language for HP 1000 RTE-based operating systems. You should be aware of which operating system you are using and on what machine the object code produced by Macro is to be executed.

Chapter 1 introduces Macro/1000, discusses backward compatibility and relocatability. It also presents some sample assembler code and lists programming aids.

Chapter 2 describes the source statement format.

Chapter 3 briefly describes all available machine instructions. This chapter should be used with the appropriate computer Operating and Reference Manual which will explain the specific machine instructions available on that machine.

Chapter 4 describes all assembler instructions, commonly called pseudo operations or pseudo ops. Conditional assembly and assembly-time variables are also discussed here.

Chapter 5 describes the macro language, how to create and access macro definitions. Macro libraries are also discussed.

Also included are the following Appendices:

- Appendix A Error Messages
- Appendix B Opcodes Arranged by Group
- Appendix C Alphabetic Listing of Opcodes
- Appendix D Binary Code
- Appendix E Assembler Operations
- Appendix F Cross Reference
- Appendix G HP Character Set
- Appendix H Relocatable Records
- Appendix I Implementation Notes
- Appendix J System Assembly-Time Variables
- Appendix K System Macro Library
- Appendix L Backward Compatible Constructs



# Table of Contents

## Chapter 1 Introducing the Macro Assembler

Compatibilities . . . . .	1-2
Backward Compatibility . . . . .	1-2
Relocatability . . . . .	1-3
Machines With Microcoding Capabilities . . . . .	1-4
Programming Process . . . . .	1-4
List Output . . . . .	1-7
Symbol Table Output . . . . .	1-10
Cross Reference . . . . .	1-10
Macro Assembler Language . . . . .	1-11
Programming Aids . . . . .	1-11
Symbolic Addressing . . . . .	1-11
Program Relocation And Relocatable Spaces . . . . .	1-12
Assembly-Time Variables . . . . .	1-13
Conditional Assembly . . . . .	1-13
Multiple Modules . . . . .	1-14
INCLUDE Statement . . . . .	1-14
Listing Control . . . . .	1-14

## Chapter 2 Coding Format

The Source Statement . . . . .	2-1
Label Field . . . . .	2-3
Opcode Field . . . . .	2-4
Operand Field . . . . .	2-4
Terms . . . . .	2-5
Symbolic Terms . . . . .	2-5
Numeric Terms . . . . .	2-6
Asterisk . . . . .	2-6
Assembly-Time Variables . . . . .	2-7
Literals . . . . .	2-8
Expressions . . . . .	2-10
Operator Precedence . . . . .	2-10
Absolute and Relocatable Expressions . . . . .	2-10
Legal Uses of Expressions . . . . .	2-11
Comment Field . . . . .	2-12
Indirect Addressing Indicator . . . . .	2-13
Statement Length . . . . .	2-14
Statement Continuation . . . . .	2-14

### Chapter 3 Machine Instructions

Memory Reference . . . . .	3-2
Word, Byte And Bit Processing . . . . .	3-4
Register Reference . . . . .	3-7
Shift-Rotate Group . . . . .	3-7
Alter-Skip Group . . . . .	3-10
Index Register Group . . . . .	3-11
No-Operation Instruction . . . . .	3-13
Extended Arithmetic Group (EAG) . . . . .	3-13
Input/Output, Overflow, And Halt . . . . .	3-15
Floating Point . . . . .	3-18
Dynamic Mapping System Instructions . . . . .	3-19
HP 1000 Fence Registers . . . . .	3-23
HP 1000 M, E, F-Series Instruction Replacements . . . . .	3-24
Replacement Formats . . . . .	3-25

### Chapter 4 Assembler Instructions

Assembler Control . . . . .	4-2
NAM . . . . .	4-4
ORG . . . . .	4-9
RELOC . . . . .	4-11
ORR . . . . .	4-12
END . . . . .	4-13
Multiple Modules . . . . .	4-13
INCLUDE . . . . .	4-15
Loader And Generator Control . . . . .	4-17
LOD . . . . .	4-17
GEN . . . . .	4-18
Program Linkage . . . . .	4-19
ENT AND EXT . . . . .	4-19
Alias . . . . .	4-20
ALLOC . . . . .	4-21
RPL . . . . .	4-23
Assembly Listing Control . . . . .	4-25
COL . . . . .	4-25
HED . . . . .	4-26
SUBHEAD . . . . .	4-27
LIST . . . . .	4-28
SKP . . . . .	4-31
SPC . . . . .	4-32
SUP . . . . .	4-33
UNS . . . . .	4-34
Storage Allocation . . . . .	4-35
BSS . . . . .	4-35
MSEG . . . . .	4-36

Constant Definition . . . . .	4-37
ASC . . . . .	4-37
BYT . . . . .	4-39
DEC . . . . .	4-40
Integer Numbers . . . . .	4-40
Floating-Point Numbers . . . . .	4-40
DEX . . . . .	4-41
DEY . . . . .	4-41
LIT . . . . .	4-42
LITF . . . . .	4-43
OCT . . . . .	4-44
Address And Symbol Definition . . . . .	4-45
DEF . . . . .	4-45
DDEF . . . . .	4-47
ABS . . . . .	4-48
EQU . . . . .	4-49
DBL AND DBR . . . . .	4-51
Declaring Assembly-Time Variables . . . . .	4-54
Substituting Values For Assembly-Time Variables . . . . .	4-54
ILOCAL, IGLOBAL, CLOCAL, CGLOBAL . . . . .	4-55
ISET, CSET . . . . .	4-58
Expressions Using Assembly-Time Variables . . . . .	4-59
Unary Operators . . . . .	4-59
Arithmetic Operators . . . . .	4-63
Comparison Operators . . . . .	4-65
Logical Operators . . . . .	4-66
Concatenation . . . . .	4-66
Conditional Assembly . . . . .	4-68
AIF, AELSEIF and AWHILE Operands . . . . .	4-68
Using AIF And AELSEIF . . . . .	4-70
Using AWHILE . . . . .	4-72
Using REPEAT And ENDREP . . . . .	4-73
Using MNOTE . . . . .	4-74



**Chapter 5 Using Macros**

Example Of a Macro . . . . .	5-2
Calling Macros . . . . .	5-3
Using Macro Libraries . . . . .	5-4
Writing Macro Definitions . . . . .	5-5
The MACRO Statement . . . . .	5-5
The Macro Name Statement . . . . .	5-6
Redefinition of Opcodes . . . . .	5-7
The Macro Body . . . . .	5-8
Comments . . . . .	5-10
The ENDMAC Statement . . . . .	5-10
Macro Parameters . . . . .	5-11
Formal Macro Parameters . . . . .	5-11
Actual Macro Parameters . . . . .	5-13
Default Parameters . . . . .	5-15



Nested Macros . . . . .	5-16
Recursion . . . . .	5-17
Creating Macro Libraries . . . . .	5-19

**Appendix A Assembler Error Messages**

**Appendix B Summary of Instructions**

Machine Instructions . . . . .	B-2
Memory Reference Instructions . . . . .	B-2
Word, Byte And Bit Processing . . . . .	B-3
No-Operation . . . . .	B-3
Register Reference, Shift/Rotate Group . . . . .	B-4
Register Reference, Alter/Skip Group . . . . .	B-5
Extended Instruction Group (Index Register Manipulation)	B-7
Input/Output, Overflow And Halt . . . . .	B-9
Extended Arithmetic Unit . . . . .	B-10
Floating-Point Instructions . . . . .	B-11
Dynamic Mapping System . . . . .	B-12
Pseudo Operations . . . . .	B-14
Assembler Control . . . . .	B-14
Loader And Generator Control . . . . .	B-14
Program Linkage . . . . .	B-15
Listing Control . . . . .	B-15
Storage Allocation . . . . .	B-16
Constant Definition . . . . .	B-16
Address And Symbol Definition . . . . .	B-16
Assembly-Time Variable Declaration . . . . .	B-17
Conditional Assembly . . . . .	B-17
Macro Definition . . . . .	B-18
Error Reporting . . . . .	B-18

**Appendix C Instructions Index**

**Appendix D Binary Codes**

**Appendix E Macro Assembler Operations**

MACRO Control Statement . . . . .	E-2
Run String Parameters . . . . .	E-4
Source . . . . .	E-4
List . . . . .	E-5
Destination . . . . .	E-6

Lines/Page . . . . .	E-7
Options . . . . .	E-7
&.RS1, &.RS2 or Work . . . . .	E-8
&.RS1, or &.RS2 . . . . .	E-8
Work . . . . .	E-8
Messages During Assembly . . . . .	E-11
On-Line Loading Of The Macro Assembler . . . . .	E-13

**Appendix F Cross Reference Table Generator**

**Appendix G HP-Character Set**

**Appendix H Relocatable Record Formats**

**Appendix I Implementation Notes**

**Appendix J System Assembly Time Variables**

&.Q . . . . .	J-1
&.ERROR . . . . .	J-2
&.RS1 AND &.RS2 . . . . .	J-3
&.REP . . . . .	J-3
&.PCOUNT . . . . .	J-4

**Appendix K HP System Macro Library**

What The System Macros Do . . . . .	K-1
A Macro Example . . . . .	K-3
Descriptions Of System Macros . . . . .	K-5
Subroutine Operations . . . . .	K-5
Macro ENTRY: . . . . .	K-5
Macro EXIT: . . . . .	K-6
Macro CALL: . . . . .	K-7
Runtime Conditionals . . . . .	K-8
Macro IF . . . . .	K-8
Macro ELSE . . . . .	K-10
Macro ELSEIF . . . . .	K-11
Macro ENDIF . . . . .	K-12
Arithmetic Operations . . . . .	K-13
Macro ADD . . . . .	K-13
Macro SUBTRACT . . . . .	K-14

Macro MAX . . . . .	K-15
Macro MIN . . . . .	K-16
Bit Operations . . . . .	K-17
Macro SETBIT . . . . .	K-17
Macro CLEARBIT . . . . .	K-18
Macro TESTBIT . . . . .	K-18
Macro FIELD . . . . .	K-20
Shifts . . . . .	K-21
Macro ROTATE . . . . .	K-21
Macro ASHIFT . . . . .	K-22
Macro LSHIFT . . . . .	K-23
Macro RESOLVE . . . . .	K-24
Text Definition . . . . .	K-25
Macro TEXT . . . . .	K-25
Macro MESSAGE . . . . .	K-25
Communication With RTE . . . . .	K-26
Macro TYPE . . . . .	K-26
Macro STOP . . . . .	K-26

## Appendix L Backward Compatible Constructs

Assembler Control Statement . . . . .	L-1
Indirection Indicator . . . . .	L-2
Clear Flag Indicator . . . . .	L-2
Old Literal Constructs . . . . .	L-2
Old Pseudo Ops . . . . .	L-3
ORB . . . . .	L-3
ORR . . . . .	L-4
IFN, IFX, And XIF . . . . .	L-5
REP . . . . .	L-7
COM . . . . .	L-8
EMA . . . . .	L-10
UNL . . . . .	L-13
LST . . . . .	L-13
MIC . . . . .	L-14
RAM . . . . .	L-15

# Chapter 1

## Introducing the Macro Assembler

Macro/1000 permits you to use all supported machine instructions for HP 1000 Computers. The Macro Assembler (MACRO) translates symbolic source language into machine code for execution on the computer. The source language provides mnemonic operation codes, assembler-directing pseudo instructions, and symbolic addressing. The assembled program can be absolute or relocatable.

Macro/1000 provides for macro calls and macro definitions. A macro definition associates a name with a group of assembler statements. When the assembler reaches a macro call statement, it expands the macro, replacing it with the source statements of the macro definition.

Why use macros? You can write a macro definition to perform a redundant section of code. The macro definitions can be general enough to perform a section of code with many different variables, both integer and string. An example of this would be a macro to generate the EXEC calling sequence. In the source code, just the macro call statement would appear, not the entire EXEC call. Another application would be to have several programs use the same macro. If the code required to perform the macro changes, then only the macro needs to be changed and the modules reassembled.

The source code can be assembled as a complete entity or it can be subdivided into several relocatable subroutines (or a main program and several subroutines). They can be assembled separately or all together in the same source file.

MACRO can read the source input from a disc file or an input device. The resultant relocatable or absolute object program is output to a disc file or an output device.

Absolute code can be loaded by the Bootstrap Loader. There are no intermediate steps needed to prepare the code before it is executed.

## Compatibilities

### Backward Compatibility

Macro/1000 has a control statement option that will provide complete backward compatibility with HP ASMB Assembly Language. You can specify ASMB in the control statement, or you can specify MACRO. Macro acts differently depending on what you specify.

If you specify ASMB in the control statement, Macro behaves in the same manner as the ASMB Assembly Language. Macro defining abilities are available, but because the ASMB function of Macro does not recognize "&-variables" as assembly-time variables or macro parameters, the usefulness of macros is limited.

If MACRO is specified in the control statement, Macro then behaves as shown in this manual. Macro produces extended relocatable records. If you have code written in ASMB and wish to run Macro with MACRO specified in the control string, be aware that Macro reserves some characters for special purposes:

- A - Macro assigns to A the value 0 (A EQU 0)
- B - Macro assigns to B the value 1 (B EQU 1)
- / - (slash) divide
- & - (ampersand) designates the start of an assembly-time variable ATV or macro parameter.
- :
- (colon) designates an attribute
- \ - (back slash) line continuation
- [,] - (brackets) designates an assembly-time array
- =,<,> - (equal, greater than, less than) used as comparison operators.
- ' - (single quote) designates a character string
- @ - (at-sign) designates indirect addressing

## Introducing The Macro Assembler

The entire instruction set of HP ASMB Assembly Language is supported on Macro/1000; however, some of the Macro instructions supersede the Assembler instructions. Appendix L (Backward Compatible Constructs) of this manual explains these instructions.

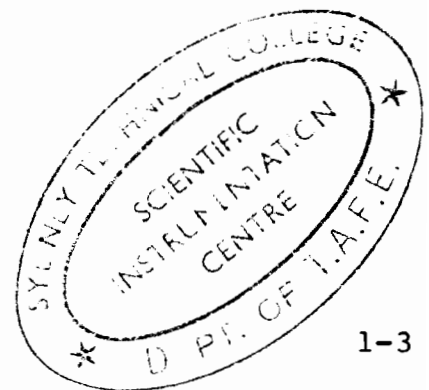
### Relocatability

The Macro Assembler produces code in a form that is ready to be relocated. This form is made up of extended relocatable records. The name "extended record" comes from the fact that EXT and ENT names may have up to 16 characters plus the fact that Macro produces some relocatable records that ASMB does not. Appendix H defines the format of all relocatable records.

The loader or generator that you use must be able to accept extended records or you must convert the extended records into non-extended relocatable format. Non-extended relocatable records can be produced by using OLDRE, a program that truncates extended records and flags incompatible records. Schedule OLDRE independently of Macro. Refer to your Utilities manual for details about OLDRE.

#### NOTE

The relocatable records produced by Macro are compatible only with RTE loaders that accept extended relocatable records. Use of these records with other loaders or any RTE generator will cause unpredictable results. OLDRE must be executed with the file containing extended records before they can be loaded on loaders that do not accept extended records, or generated using any RTE generator.



## Machines with Microcoding Capabilities

Some HP 1000 machines have software equivalents for instructions that are implemented in microcode in others. For example, the instructions using the X- and Y-Registers are microcoded in the M-, E-, and F-Series machines. On the L- and XL-Series, there is no microcode for these instructions, so they have been coded in software.

For the instruction "CAX" (copy A to X), Macro will generate a jump to a location external to the program. When the relocatable code is loaded, the loader determines whether or not there is microcode for the instruction and replaces the jump with a microcode instruction or a jump to the software routine. Appendix C of this manual has information on which instructions have software equivalents on a particular machine. Refer to the Operating and Reference Manual for your HP 1000 Series machine for more information.

Macro/1000 will replace an instruction with microcode if requested. The 'I' option in the control statement (discussed in Appendix E) will cause macro to generate microcode replacements for these instructions. You may specify the 'I' option if your machine has microcoding.

## Programming Process

The programming process consists of creating a source file, assembling the source file to produce relocatable code, loading the relocatable code, and then executing the program. A sample source file is shown in Figure 1-1. This file is a simple routine which counts the number of ones in the A-Register. Note that the source code of a module must have the following statement, depending on whether the module is relocatable or absolute:

<u>Relocatable</u>	<u>Absolute</u>
Control statement	Control statement
NAM statement	ORG statement
END statement	END statement

## Introducing The Macro Assembler

The control statement is the first statement which contains a set of options. In this example, the R (relocatable source), L (output to a list file), and T (list symbol table) options have been chosen. More information on the control statement is found in Appendix E.

The NAM statement immediately follows the control statement (except for comments, a HED or SUBHEAD statement, macro definition, or conditional assembly). The NAM statement indicates the origin of a relocatable program; an ORG statement indicates the origin of an absolute program.

The END statement is the last statement of the module and may contain a transfer address for the start of a relocatable program. After the END statement, however, can be conditional-assembly or other statements that do not produce code. There can also be another module or NAM-END pair.

```
MACRO,R,L,T
    NAM COUNT
    ENT COUNT
;
; Subroutine to count the no. of set bits in the
; A-Register
;
COUNT    NOP           ; subroutine entry point.
          CLB           ; clear B-Register (B used to
          ; count # of 1's).
          LDX =D16      ; load 16 into X-Register.
repeat    SLA           ; skip if bit 0 of A-Reg is on.
          INB           ; yes, add 1 to count in B.
          RAL           ; rotate A-Register left 1.
          DSX           ; decrement X, skip if 0, done?
          JMP repeat    ; not done, repeat.
          JMP @COUNT   ; return to main program.
          ; number of 1's in B-Register.
END
```

Figure 1-1. Source Code Example



## Introducing The Macro Assembler

After you create a file that has Macro Assembler (MACRO) source statements, it is ready to be assembled. MACRO assembles your file by doing the following:

1. Expand macros.
2. Check for syntax errors in the source statements.
3. Create the list file.
4. Create relocatable code.

The relocatable code produced by MACRO is then ready to be loaded using the Loader program. The Loader produces memory image code which the computer can execute. The whole process is illustrated in Figure 1-2. On RTE-A the Loader program is called LINK.

In the process, the Macro Assembler also produces a listing of the code as well as a symbol table. These listings are explained below.

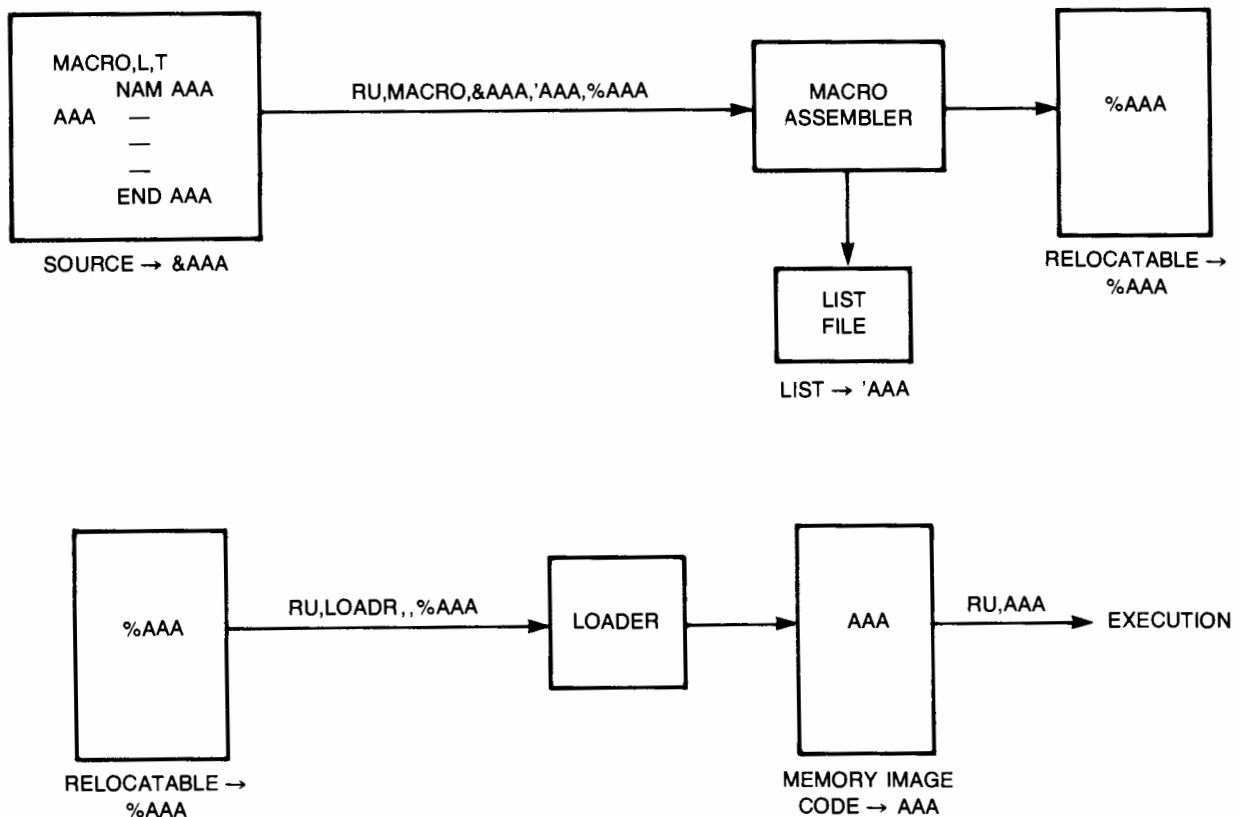


Figure 1-2. Assembly Process

## List Output

Figure 1-3 shows the assembled listing of the sample code. The header contains a sequential page number and time of day information. Figure 1-4 defines the fields in the listing, using lines 12 and 17 for illustration.

The relocation or external symbols that indicate the type of relocation to be done for the operand field are as follows:

<u>CHARACTER</u>	<u>RELOCATION BASE</u>
Blank	Absolute
R	Program relocatable
C	Common relocatable
X	External symbol
B	Base page relocatable
S	Substitution code
E	Extended Memory Area
V	SAVE relocatable area

A plus (+) in column 21 indicates the code came from a macro expansion. A minus (-) marks code that appeared in conditional assembly statements that did not get assembled. The last section of this chapter has a brief paragraph about conditional assembly.

Lines consisting entirely of comments using a semicolon (;) in column 1 show the source statement sequence number in the first five columns and the comment beginning in column 22.

Lines consisting entirely of comments using an asterisk (\*) in column 1 show the statement number in the first five columns and the comment beginning in column 7.

# Introducing The Macro Assembler

```

PAGE# 1      Macro/1000 Version 1.0  9:55 AM MON., 4 AUG., 1980

00001          MACRO,R,L,T
00002          NAM COUNT
00003          ENT COUNT
00004          ;
00005          ; subroutine to count the no. of set bits
00006          ; in the A-Register
00007          ;
00008 00000 000000 COUNT  NOP          ;subroutine entry point.
00009 00001 006400      CLB          ;clear B-Reg (B used to
00010          ;count # of 1's).
00011 00002 014001X    LDX =D16     ;load 16 into X-Reg
          00003 000012R
00012 00004 000010 repeat SLA      ;skip if bit 0 of A-Reg is 0.
00013 00005 006004      INB          ;on yes, add 1 to count in B.
00014 00006 001200      RAL          ;Rotate A-Reg left 1.
00015 00007 014002X    DSX          ;decrement X, skip if 0,done?
00016 00010 024004R    JMP repeat  ;not done, repeat.
00017 00011 124000R    JMP @COUNT ;return to main program.
00018          ;number of 1's in B-Reg.
          00012 000020      END
  
```

Macro: No errors total

Figure 1-3. Assembled Listing Of Sample Code

```

00012 00004 000010 repeat SLA          ;skip if bit 0 of A-Reg is 0.
00017 00011 124000R          JMP @COUNT ;return to main program.
  
```

↑ source statement sequence number  
 ↑ location in octal  
 ↑ object code in octal  
 ↑ relocation or external symbol  
 ↑ label field  
 ↑ operation (op) code  
 ↑ operand field  
 ↑ comments

Figure 1-4. Listing Fields

## Introducing The Macro Assembler

For each error found in the source code, MACRO prints an error message. A caret (^) points to the location at which MACRO found the error. Immediately after the error is the following message:

```
nnn >> <text>
```

and at the end of the code:

```
ERROR nnn in line LLL <macro line # mmm> <Include file #iii>
```

where:

nnn is the error number.

<text> is an explanation of the error.

LLL is the line number where the error occurred.

mmm is the line number inside of a macro definition where the error occurred. This phrase is printed only if an error occurred inside the macro.

iii is the file number of the included file. This phrase is printed only if an error occurred inside of the include file. In this case LLL is the included file's line number.

The format of the error messages makes locating them very easy. You can scan the list file for " >> " (using the Editor) to show any error messages. Knowing the line numbers of the errors (given at the end of the listing), you can find the specific errors.

## Symbol Table Output

Figure 1-5 shows the symbol table listing produced when the example source code was assembled. A symbol table contains all of the symbols and their relocation type created during the assembly in alphabetic order. Columns 8 through 23 contain the name of the label. Columns 34 through 39 contain the value of the label. Column 40 specifies the type of relocation for the operand field.

.DSX	000002X	(External Symbol ID#)
.LDX	000001X	" " "
A	000000	(Absolute memory location)
B	000001	" " "
COUNT	000000R	(Relocatable memory reference)
REPEAT	000004R	" " "

Figure 1-5. Sample Symbol Table Listing

## Cross Reference

The cross-reference table generator is useful for larger programs. Not only are the symbols defining addresses given, but also the addresses where the symbols are used or changed. To have the cross-reference table listed after the assembly, specify the 'C' option in the control statement. Appendix F of this manual details the output of the cross-reference table generator.

## Macro Assembler Language

The Macro Assembler language consists of the following opcodes:

1. Machine instructions, which instruct the machine to do something such as manipulate registers or send flags to the operating system.
2. Assembler instructions, which instruct the Macro Assembler to do something such as create space for a value or specify a listing option.
3. Macro calls which cause the code of a macro definition to be generated at that point in the code.

## Programming Aids

Macro/1000 provides many tools to aid the programmer:

### Symbolic Addressing

A symbol represents the address for a word in memory. A symbol is defined when it is used as:

- \* A label for a location in the program;
- \* A name of a common storage area;
- \* The label of a data storage area or constant;
- \* The label of an absolute or relocatable value;
- \* A location external to the program;

Through use of arithmetic operations, symbols can be combined with other symbols or numbers to form an expression that can identify another location in memory. Symbols that appear in operand field expressions but are not defined and symbols that are defined more than once are flagged as errors by the assembler.

## Program Relocation and Relocatable Spaces

Relocatable records produced by Macro are assigned absolute addresses by the loader. The assembler assumes a starting location of 0 for relocatable code. This is called the relative origin. The loader determines the absolute origin of the code and then relocates the remainder of it with respect to its absolute origin. In other words, the value of the absolute origin is added to each relocatable address to produce the absolute address.

Macro/1000 has five different types of relocatable spaces:

- program relocatable
- base page relocatable
- EMA relocatable
- SAVE relocatable
- common relocatable

Each space has its own relative origin. Also, each space has its own counter. A counter assigns consecutive memory addresses to source statements within its relocatable space.

For example, source statements in the main portion of a block of code will be in the program relocatable space. The assembler assigns the first statement to be the program relative origin and maintains the block of code with the program location counter.

The initial value of the program location counter is established according to the use of either the NAM or ORG pseudo operation at the start of the program. The NAM operation causes the program location counter to be set to zero for a relocatable program; the ORG operation specifies the absolute starting location for an absolute program.

A relocatable program may specify that certain operations or data areas be allocated to different relocatable spaces. For example, through the RELOC command, a data area is specified to be in the common relocatable space. That common area has its own relative origin and is maintained by the common location counter.

Another type of memory space to be considered is "absolute" space. This refers to a program and its data which is loaded directly into memory for sole occupancy of the HP 1000. This is usually accomplished by a bootstrap loader taking the code directly from a device such as cassette tape or standard magnetic tape. These programs must load and run entirely on their own; that is, no external address fix-up is done by the loader, and no program relocation is done by the operating system.

Therefore all addresses which you set up in your program are absolute and fixed, hence the term "absolute program".

A common example of an absolute program is !BCKUP, an offline backup and restore utility program which is distributed by Hewlett-Packard with the RTE-6/VM operating system.

### Assembly-Time Variables

Assembly-time variables (ATVs) are variables whose values are defined, manipulated and used at assembly-time. Therefore, they do not take up space in relocatable code. As the source file is being assembled, the current value of the ATV is substituted into the code. For example:

```
&P1  IGLOBAL  0      ; set &P1 to 0.
      REPEAT   5      ; Start REPEAT loop.
      DEC &P1
&P1  ISET &P1+1    ; alter the values of &P1.
      ENDREP
```

will generate:

```
DEC 0
DEC 1
DEC 2
DEC 3
DEC 4
```

ATVs can be used as flags and counters which direct the assembler in processing the user's program.

The value assigned to an assembly-time variable may take on one of two types: integer or character. They may be of local scope (local only to a macro definition, REPEAT or AWHILE loop) or global scope (global to the entire program).

### Conditional Assembly

Conditional assembly allows you, along with using assembly-time variables, to assemble only certain portions of your program. For example, suppose a program has an error reporting section that does not need to be assembled all the time. You can designate an ATV as a flag; then, depending on the value of the flag, the error reporting section may or may not be assembled.



## Multiple Modules

The ability to have more than one module in one file means that a main routine and its subroutines can be assembled and loaded together. Combining this concept with conditional assembly gives the option of assembling only certain modules.

## INCLUDE Statement

The INCLUDE statement causes the assembler to continue assembling from the source code file specified in the operand field. When MACRO encounters the INCLUDE statement, it begins the line numbering for the listing at line number one again. When the end of this file is reached, assembly continues at the statement following the INCLUDE in the original file.

## Listing Control

Macro/1000 has a full set of listing control pseudo operators. Among the pseudo ops are commands to suppress the listing of macro expansions, suppress additional code line listings, skip to the top of the next page, or specify a heading or subheading. The pseudo op LIST has a keyword parameter to do many of the above options. Another pseudo op, COL, controls what columns in which the mnemonic, operand, and comments start in the assembled listing.

# Chapter 2

## Coding Format

The source code for an assembly language program consists of a series of source statements. The format of the source statements is described in this chapter. First, the parts of a source statement are introduced. Then each part, or field, is discussed in detail. Finally, the methods used to combine these fields into valid source statements are presented.

### The Source Statement

A statement within a Macro/1000 source program can contain a maximum of four parts known as fields: the label field, opcode field, operand field and comment field.

Other than the label field, which must begin in the first column of the statement, the column in which a field begins is not important, except that column one must be blank if there is no label. However, the fields used in a statement must appear in the following order.

1. label
2. opcode
3. operand
4. comment

Separate the label, opcode and operand fields by at least one space. Separate the comment field from the other fields by a semicolon.

The label field allows a statement to be associated with a symbolic name. Labels are optional. If a label is used in a statement, this statement can then be accessed by other statements within the program. For example, a section of code which processes errors encountered by the program might begin with the statement:

```
error.process    cpa    bit.field
```

## Coding Format

By assigning the label 'error.process' to the statement, other statements can access this section of code by referring to that label.

The opcode field holds mnemonic groups of characters that describe actions to be performed. For example, the statement:

```
cla
```

clears the A-Register.

The operand field provides information required by an opcode to complete its action. In the statement:

```
jmp error.process
```

the opcode field contains the opcode 'jmp' which tells the program to continue processing the statements at the location specified by the following operand field. In this case, the location is specified by the label name 'error.process'.

The comment field is an optional field you can use to clarify the meaning of a statement or a section of source code. Identify the comment field by an asterisk (\*) in the first column or by a semicolon (;) elsewhere in the statement. The asterisk denotes the entire statement as a comment; the semicolon denotes all remaining characters in the statement as a comment. For example, if the statement:

```
cla
```

were contained in an obscure piece of source code, the reader might understand what the command does, but not why it was used. Comments should be used in the source code to explain this. In the previous example, comments such as:

```
* Begin section of code which determines number of iterations  
*****
```

```
cla ; Prepare A-Register as counter.
```

would make the statement easier to understand.

## Label Field

The optional label field identifies the statement. It is used as a reference point by other statements in the program which need to access the contents of the location represented by the label.

The label field starts in column one of the statement and is terminated by at least one space.

A label can have one to sixteen characters. The starting character can be any of the following:

A-Z	
a-z	- mapped to upper case
!	- exclamation point
"	- double quote
\$	- dollar sign
%	- percent sign
^	- up carat
?	- question mark
.	- period
#	- pound sign
{ }	- braces
_	- underscore

The next 15 characters in the label may be any of the starting characters, the digits 0-9, or the "at" sign(@).

If you enter a label of more than sixteen characters, the Macro Assembler will flag this condition as an error.

Examples of legal labels:

```
check.overflow
idsegment
?LISTFLAG
A!"$%^9.
```

Examples of illegal labels:

3abcd	starts with a number
abcdefghijklmnopq	greater than 16 characters
@idmem	character '@' is not allowed as starting character

## Coding Format

The Macro Assembler defines the labels A and B for you. They have absolute values of 0 and 1, respectively. You cannot redefine them.

### Opcode Field

An opcode is a group of characters which specifies an action to be performed by the assembler. An opcode may be a machine instruction, an assembler instruction, or a macro call.

Machine instructions are commands to the assembler to put a binary machine instruction into a memory location. They are discussed briefly in Chapter 3 and are discussed in detail in the Operating and Reference Manual for your computer.

Assembler Instructions are commands to the assembler to fill memory locations with octal values. They are generally referred to as pseudo operations and are discussed in Chapter 4.

A macro call tells the assembler to substitute a specified set of machine instructions and pseudo operations. For more details refer to Chapter 5.

The opcode field follows the label field and is separated from it by at least one space. If there is no label, the opcode can begin anywhere after column one.

### Operand Field

The meaning and format of the operand field depends upon the type of opcode used in the source statement. An operand can be a single value (term) or it can contain a combination of these, joined by operators (an expression).

The operand field follows the opcode field separated by at least one space.

Examples:

```
JMP found.error ;transfer control to location 'found.error'.  
JMP found.error+5 ;transfer control to 5 locations past  
;'found.error'.  
LDA 1717B ;load register A with contents of memory  
;location 1717 octal.
```

## Terms

Terms appear in the operand field of a source statement and are used in the source program to represent values. In Macro/1000 there are several types of terms: symbolic terms, numeric terms, the asterisk, assembly-time variables and literals.

### Symbolic Terms

A symbolic term must be a symbol that is defined elsewhere in one of the following ways:

- \* As a label in the label field of a machine instruction or a macro call.
- \* As a label in the label field of a BSS, ASC, DEC, DEX, OCT, DEF, DDEF, BYT, ABS, EQU, DBL, or DBR assembler instruction.
- \* As a name in the operand field of an EXT assembler instruction.

The value of a symbolic term is absolute or relocatable depending on the assembly option you select. Macro assigns a value to a symbol as it appears in one of the above fields of a statement. If a program is to be loaded in absolute form, the values assigned by Macro remain fixed. If the program is to be relocated, the actual value of a label is established on loading. A symbol may be assigned an absolute value through use of the EQU pseudo instruction.

A symbolic term may be preceded by a minus sign. If preceded by a plus or no sign, the symbol refers to its associated value. If preceded by a minus sign, the symbol refers to the two's complement of its associated value.

Examples:

```
LDA  A1234          ;valid operand
ADA  B.1           ;valid operand
JMP  ENTRY         ;valid operand
```

## Coding Format

### Numeric Terms

A numeric term may be a decimal or octal integer. A decimal number is represented by one to five digits within the range -32768 to 32767. An octal number is represented by one to six digits followed the letter B (0 to 177777B).

For a memory reference instruction in an absolute program, the maximum value of a numeric operand depends on the type of machine instruction or pseudo operation. Numeric operands are absolute. Their value is not altered by the Assembler or the Loader.

### Examples:

```
MAX DEC 32767 ; define maximum
TBL BSS 100   ; reserve array
WCS EQU 10B   ; define I/O select code
```

### Asterisk

An asterisk (\*) that appears in the operand field alone or next to an arithmetic operator refers to the value in the current location counter at the time the source program statement is encountered.

If assembly is taking place in the program relocatable space, then an asterisk refers to the program relocatable counter. If in the base page space, then an asterisk refers to the base page counter, and so on. If the asterisk appears in between two numeric or symbolic terms, then it is interpreted as the multiplication operator.

### Example:

```
JSB EXEC
DEF *+2 ; location of return
DEF =D6
```

## Coding Format

### Assembly-Time Variables

Assembly-time variables (ATVs) are variables whose values are defined, manipulated, and used at assembly time. There is no space allocated for their values in the object code; their values are known only to the assembler as it processes the source program. The assembler scans each line of source code and substitutes the value of any assembly-time variable occurring outside of the comment field.

Assembly-time variable names can be from 1 to 16 characters long. The first character must always be an ampersand (&). The next characters, if present, can be any combination of letters (A-Z or a-z - lower case mapped to upper), and digits.

The character period (.) is legal anywhere after the ampersand (&) but, by convention, system assembly-time variables begin with the character sequence "&.". The assembler will not mark an error if you declare the assembly-time variables starting with "&.". However, if you declare a variable that is the same as a system variable, an error will result. To insure future compatibility, you are strongly encouraged not to declare assembly-time variables starting with "&.".

The value assigned to an assembly-time variable can be type integer or type character. A type integer assembly-time variable has a value ranging from -32768 to +32767, while a type character consists of from 0 to 80 ASCII characters.

Refer to Chapter 4 for information on declaring assembly-time variables and changing their values.



## Literals

Literal values can be specified as operands in relocatable or absolute programs. The assembler converts the literal to its binary value, assigns an address to it, and substitutes this address as the operand. Locations assigned to literals are those immediately following the last location used by the module, or by locations immediately following usage of the LIT or LITF command.

To specify a literal, use an equal sign and a one-character identifier defining the type of literal. Specify the actual literal value immediately following this identifier; no spaces may intervene.

The identifiers are:

- =D A decimal integer, in the range -32768 to 32767.
- =F A floating point number; any positive or negative real number in the range  $10^{*-38}$  to  $10^{*38}$ .
- =B An octal integer, one to six digits between 0 and 7, resulting in an octal value between 0 and 177777B.
- =L An expression which, when evaluated, will result in an absolute, external, or single word relocatable value. All symbols appearing in the expression must be defined before they are used with this construct.
- =S A string surrounded by single quotes.
- =R A right-justified, zero-filled ASCII character.

If you use the same literal in more than one instruction or if different literals have the same value (e.g., =B100 and =D64), only one value is generated, and all instructions using these literals refer to the same location.

## Coding Format

Literals can be specified only in the following memory reference, register reference, EAU instructions, and pseudo operations:

ADA	CPB	LDX
ADB	DEF	LDY
ADX	DDEF	MBT
ADY	DIV	MPY
AND	IOR	MVW
CBS	JRS	SBS
CBT	LDA	TBS
CMW	LDB	XOR
CPA		

This group may use  
=D    =B    =L    =S    =R



DLD	FDV	FSB
FMP	FAD	DEF

This group may use  
=F

### Examples:

LDA	=D7980	; A-Register is loaded with the binary ; equivalent of 7980.
IOR	=B777	; Inclusive OR is performed with the ; contents of A-Register and 777B.
LDB	=LZETZ-ZOOM+68	; B-Register is loaded with the absolute ; value resulting from the given ; expression.
LDA	=LARRAY	; Load A-Register with the address of ; ARRAY.
FMP	=F39.75	; Contents of the A- and B-Register get ; multiplied by 39.75.
STR DEF	=S'long string'	; Address of string put into memory.
MIN DEF	=D-32768	; Address of smallest decimal integer is ; put in memory.
CHR DEF	=RA	; The lower byte of CHR contains an A, ; the upper byte is zero-filled.

## Expressions

An expression is a combination of terms and operators that can be resolved to a value. There are several types of operators that can be used to form arithmetic expressions in Macro/1000.

unary operator	-	(negate)
arithmetic operators	*	(multiply)
	/	(divide)
	+	(add)
	-	(subtract)

### Operator Precedence

The evaluation of expressions is performed from left to right in the statement.

Only the unary operations and operations within parentheses are performed with a higher precedence than any other operations.

### Absolute and Relocatable Expressions

An expression is absolute if its value is unaffected by program relocation. An expression is relocatable if its value changes according to the location in which the program is loaded.

In an absolute program, all expressions are absolute. In a relocatable program, an expression may be program relocatable, common relocatable, base page relocatable, or absolute, depending on the definition of the terms, and the operators composing it.

If both terms on an expression of the form:

T1 operator T2

are absolute, the result of the expression will also be absolute. If one term is relocatable and the other is absolute, the result will be a relocatable term.

### Legal Uses of Expressions

Although expressions are legal for Memory Reference Instructions, Extended Arithmetic, Floating Point, Memory Expansion, DBL, DBR and DEF pseudo ops, users should only use one address per expression. Subsequent elements of the expression should consist only of absolute terms. For example, a legal expression would be:

```
LDA addr+1
```

The term "addr" is a relocatable address while the 1 is absolute.

An example of an illegal expression is:

```
LDA addr1 + addr2
```

This expression is not allowed because it contains two relocatable elements.

## Comment Field

The comment field allows you to transcribe notes on the program that will be listed with source language coding on the output produced by the Assembler.

The semicolon is a comment delimiter. In some places it is required, but is optional as a starting character on most comments. If a semicolon appears as the first non-blank character on a line, the entire line is taken as a comment. The opcodes on which it is required are:

END (required only if the entry point is not specified)

AIF, AWHILE, REPEAT, and all macro calls

IGLOBAL, CGLOBAL, ILOCAL, CLOCAL, ISET, and CSET

HLT (required when no select code is given)

MIC instruction calls

An asterisk (\*) appearing as the first character on a line, will also denote the entire line as a comment.

On the list output, statements consisting entirely of comments started by a semicolon begin in column 22. A comment starting with an asterisk in column one starts in column 8 on the listing. If any statement exceeds 128 characters because of this relocation, characters beyond that limit will not appear on the listing. If any line is longer than 120 characters after string substitution occurs, an error will result.

## Indirect Addressing Indicator

The HP1000-Series computers provide a hardware indirect addressing capability for memory reference instructions. The operand portion of an indirect instruction contains the address of another location. The secondary location can be the operand or it can be indirect also and give yet another location, and so forth. The chaining ceases when a location is encountered that does not contain an indirect address.

To specify indirect addressing in Macro/1000, prefix the memory reference with an "at" sign (@). The actual address of the instruction is typically given in a DEF pseudo operation; this pseudo operation may also be used to indicate further levels of indirect addressing.

Example:

```

AB    LDA    @SAM          ; The value 10 is loaded
SAM   DEF    @ROGER       ; to the A-Register.
ROGER DEF    BOB
BOB   DEC    10

```

A relocatable assembly language program can be designed without concern for the pages in which it will be stored; indirect addressing is not required in the source language. When the program is loaded, the loader provides indirect addressing whenever it detects an operand which does not fall in the current page or the base page. The loader substitutes a reference to a program link location (established by the loader in either the base page or the current page) and then stores an indirect address in the particular program link location. If the program link location is in the base page, references to the same operand from other pages will be via the same link location.

## Statement Length

A source line may contain up to 128 characters including spaces, before a statement continuation marker is required.

If no continuation marker is found before the line exceeds 128 characters, the line is truncated without warning.

If the statement length is zero, the Macro Assembler generates a new number for that line and treats it as a comment.

## Statement Continuation

To continue a statement onto the next line, use the backslash character (\) after the last character on the line that you wish the assembler to recognize as an operand. The assembler then reads the next line to continue the statement. Any leading blanks on that line will be ignored. Anything on a line after a backslash is considered to be a comment.

The backslash is not permitted in the label or the opcode field. Line continuation is not permissible in the middle of a string, assembly-time variable name, user label, integer or array reference. If a backslash appears in a string (that is, surrounded by single quotes), it does not cause line continuation.

Example:

```
MYMACRO HAS,          \Example of
    A,                \a continuing
    LOT,              \macro call
    OF,PARAMETERS    ;statement
```

# Chapter 3

## Machine Instructions

Machine instructions are the object code generated by the Assembler. Each instruction corresponds to a mnemonic operation code (opcode) and, usually, an operand. An assembly-language program statement contains a machine instruction, and may or may not start with a label, by means of which it can be referenced from other statements in the program.

Machine instructions are briefly discussed in this chapter. Refer to the appropriate computer Operating and Reference Manual for a full description of each machine instruction.

The following notations are used in the description of machine instructions. They are also used in the remainder of this manual.

label Optional statement label.

m Memory location: an expression that evaluates to a symbolic address or that may be resolved to a symbolic address through various levels of indirection.

@ Indirect addressing indicator.

sc Select code: an expression that evaluates to an integer within the range of 0 to 63.

C Clear interrupt flag indicator.

The machine instructions are classified as follows:

- Memory Reference;
- Word, Byte and Bit Processing;
- Register Reference;
- Index Register;
- No-Operation;
- Extended Arithmetic;
- Input/Output, Overflow and Halt;
- Floating Point;
- Dynamic Mapping System.



## Memory Reference

The memory reference instructions perform arithmetic, logical, and jump operations on the contents of memory locations and the registers. Statements containing these opcodes can take one of two syntactical forms, depending on the opcode used.

Where operands are stacked vertically in the following, only one can be used.

The first form is:

$$[\text{label}] \text{ opcode } \left\{ \begin{array}{l} m \\ @m \\ \text{literal} \end{array} \right\} [;\text{comment}]$$

Opcodes that require this form are:

ADA - Add the contents of m to A.

ADB - Add the contents of m to B.

AND - Logical "and" of the operand value and the contents of A are placed in A.

CPA - Compare the value of the operand with the contents of A. If they differ, skip the next single word instruction.

CPB - Perform the same operations as CPA on the contents of the B-Register.

IOR - Inclusive "or" the operand value and the bits in A. Place the result in A.

LDA - Load A with the contents of m.

LDB - Load B with the contents of m.

XOR - Exclusive "or" the operand value and the bits in A. Place the result in A.

Only =S, =D, =B, =A, and =L literals are accepted with these opcodes.

## Machine Instructions

The second form is:

```
[label] opcode { m } [;comments]
                { @m }
```

Opcodes that require this form are:

ISZ - Increment, then skip if the result is zero.

JMP - Jump to m.

JSB - Jump to subroutine. Return to address following that stored in m. Execution proceeds at location following m. A return to the main program sequence will be effected by a JMP indirect through location m.

STA - Store contents of A in the address specified by operand.

STB - Store contents of B in the address specified by operand.

## Word, Byte and Bit Processing

The word-processing instructions move a series of data words from one array in memory to another or compare (word by word) the contents of two arrays in memory. The word-processing instructions are MVW and CMW.

The byte-processing instructions copy a data byte from memory into the A-Register, copy a series of data bytes from one array in memory to another, compare (byte-by-byte) the contents of two arrays in memory, or scan an array in memory for particular data bytes. The byte-processing instructions are LBT, SBT, MBT, CBT and SFB.

A byte address is an expression that is the symbolic address of a byte location. The address occupies 16 bits; bits 1-15 indicate the address of the word containing the byte, and bit 0 indicates a high order (bit is clear) or low order byte (bit is set).

The bit-processing instructions selectively test, set or clear bits in a memory location according to the contents of a mask. The bit-processing instructions are TBS, SBS and CBS.

Instructions in this group are implemented by calls to external subroutines when used on the L-Series systems.

The statements containing these opcodes may take one of the following three syntactical forms. One of these forms is:

$$[\text{label}] \text{ opcode } \left\{ \begin{array}{l} m \\ @m \\ \text{literal} \end{array} \right\} [;\text{comment}]$$

Opcodes that require this form are:

CBT - Compare bytes beginning at byte address in A to the bytes beginning at byte address in B. The number of bytes to be compared is indicated by the value of the operand. Comparison stops when either the first unequal byte is reached, or the number of bytes specified by operand has been compared.

## Machine Instructions

If both arrays are equal, execution proceeds at the next word following the instruction. If the array specified by A is less than the second array, execution proceeds at the second word following the instruction. If array specified by A is greater than the second array, execution proceeds at the third word following the instruction.

After execution, register A contains the address of the byte in the first array where comparison stopped, and B contains its original value, incremented by the number of bytes compared.

CMW - Compare words beginning at the address in A to the words beginning at address in B. Neither address may be indirect. Number of words to be compared is indicated by the operand value. Comparison stops when either first unequal word is reached, or number of words specified by operand has been compared.

If both arrays are equal, execution proceeds at word following instruction. If array specified by A is less than second array, execution proceeds at second word following instruction. If array specified by A is greater than second array, execution proceeds at third word following instruction.

After execution, the A-Register contains the address of the byte in the first array, where comparison stopped, and the B-Register contains the original value, incremented by the number of words compared.

MBT - Move bytes beginning at the byte address in A to the byte address in B. The operand specifies number of bytes to be moved. A and B are incremented by the number of bytes moved.

MVW - Move words beginning at the address stored in A to the address in the B. Neither address may be indirect. The operand specifies the number of words to be moved. A and B are incremented by the number of words moved.

NOTE: Refer to the pseudo ops DBL and DBR in Chapter 4 for more information on byte addressing.

## Machine Instructions

Another syntactical form is:

```
[label] opcode [;comments]
```

Opcodes that require this form are:

- LBT - Load byte from the byte address contained in B into the lowest eight bits of A, and increment B.
- SBT - Store the byte contained in the lowest eight bits of A into the byte address contained in B, and increment B.
- SFB - Scan for byte. A contains a test byte in bits 0-7 and a termination byte in bits 8-15. The beginning address of the array to be scanned is stored in B. The array is scanned until a byte matches either the test or termination byte. If a byte in the array matches the test byte, execution proceeds at the next sequential location, and B will contain the address of the byte matching the test byte.

If a byte in the array matches the termination byte, the instruction will skip one word upon exit, and B will contain the address of the byte matching the termination byte, plus one.

The third syntactical form is:

```
[label] opcode      m          m  
                  @m          @m  [;comments]  
                  literal
```

with at least one blank between operands.

Opcodes that require this form are:

- CBS - Clear the bits contained in the address of the second operand that corresponds to the bits that have been set in the value of the first operand.
- SBS - Set the bits contained in the address of the second operand that corresponds to the bits that have been set in the value of the first operand.
- TBS - Test the bits contained in the address of the of the second operand with the bit mask specified by the first operand. Only the bits that are set in the bit mask are tested. If all the bits tested are 1's, the next instruction is obeyed. Otherwise, the computer will skip one instruction.

## Register Reference

The register reference instructions are used to test and manipulate the contents of registers. These instructions can be divided into two groups, the shift-rotate group and the alter-skip group.

### Shift-Rotate Group

The shift-rotate instructions are listed and briefly described below. These instructions are illustrated in Figure 3-1.

- ALF - Rotate A left four bits.
- ALR - Shift A left one bit, clear sign, zero to least significant bit.
- ALS - Shift A left one bit, zero to least significant bit; sign unaltered.
- ARS - Shift A right one bit, extend sign; sign unaltered.
  
- BLF - Rotate B left four bits.
- BLR - Shift B left one bit, clear sign, zero to least significant bit.
- BLS - Shift B left one bit, zero to least significant bit; sign unaltered.
- BRS - Shift B right one bit, extend sign; sign unaltered.
  
- CLE - Clear E to zero.
  
- ELA - Rotate E and A left one bit.
- ELB - Rotate E and B left one bit.
- ERA - Rotate E and A right one bit.
- ERB - Rotate E and B right one bit.
  
- LAE - Copy the low-order bit of A into E; A is unchanged.
- LBE - Copy the low-order bit of B into E; B is unchanged.
  
- RAL - Rotate A left one bit.
- RAR - Rotate A right one bit.
- RBL - Rotate B left one bit.
- RBR - Rotate B right one bit.
  
- SAE - Copy the sign bit of A into E; A is unchanged.
- SBE - Copy the sign bit of B into E; B is unchanged.
- SLA - Skip the next single-word instruction if the least significant bit in A is zero.
- SLB - Skip the next single-word instruction if the least significant bit in B is zero.

# Machine Instructions

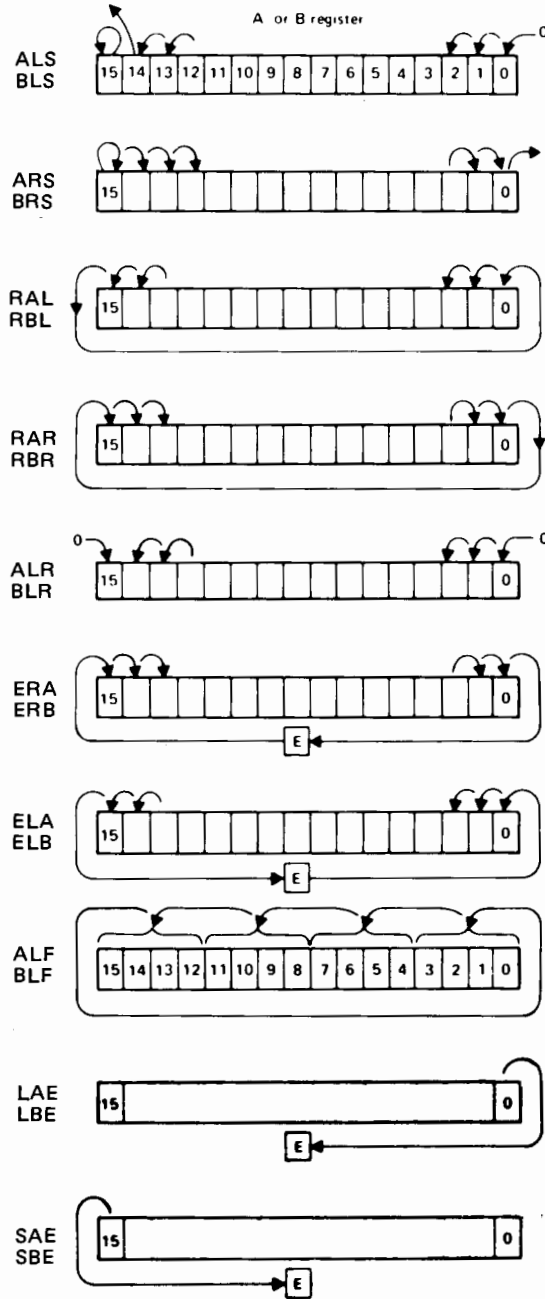


Figure 3-1. Instructions Of The Shift-Rotate Group.

## Machine Instructions

The opcodes within the shift-rotate group can be combined as follows:

$$[\text{label}] \left( \begin{array}{l} \text{ALS} \\ [\text{ARS}] \\ \text{RAL} \\ \text{RAR} \\ \text{ALR} \\ \text{ALF} \\ \text{ERA} \\ \text{ELA} \\ \text{SAE} \\ \text{LAE} \end{array} \right) [,\text{CLE}] [,\text{SLA}] \left( \begin{array}{l} ,\text{ALS} \\ [,\text{ARS}] \\ ,\text{RAL} \\ ,\text{RAR} \\ ,\text{ALR} \\ ,\text{ALF} \\ ,\text{ERA} \\ ,\text{ELA} \\ ,\text{SAE} \\ ,\text{LAE} \end{array} \right) [;\text{comments}]$$

$$[\text{label}] \left( \begin{array}{l} \text{BLS} \\ [\text{BRS}] \\ \text{RBL} \\ \text{RBR} \\ \text{BLR} \\ \text{BLF} \\ \text{ERB} \\ \text{ELB} \\ \text{SBE} \\ \text{LBE} \end{array} \right) [,\text{CLE}] [,\text{SLB}] \left( \begin{array}{l} ,\text{BLS} \\ [,\text{BRS}] \\ ,\text{RBL} \\ ,\text{RBR} \\ ,\text{BLR} \\ ,\text{BLF} \\ ,\text{ERB} \\ ,\text{ELB} \\ ,\text{SBE} \\ ,\text{LBE} \end{array} \right) [;\text{comments}]$$

Where the parameters are shown stacked, only one can be used. The brackets ([]) indicate optional parameters.

CLE, SLA, or SLB appearing alone or in any valid combination with each other are assumed to be a shift-rotate machine instruction, even though they are also in the alter-skip group.

At least one and up to four of the shift-rotate instructions are included in one statement. Instructions referring to the A-Register cannot be combined in the same statement with those referring to the B-Register.



**Alter-Skip Group**

The instructions in the alter-skip group are:

- CCA - Clear, then complement A (set to ones).
- CCB - Clear, then complement B (set to ones).
- CCE - Clear, then complement E.
- CLA - Clear A.
- CLB - Clear B.
- CLE - Clear E.
- CMA - Complement A.
- CMB - Complement B.
- CME - Complement E.
  
- INA - Increment A by one.
- INB - Increment B by one.
  
- RSS - Reverse the sense of the skip instruction; if no skip instruction precedes RSS in the statement, skip the next instruction.
  
- SEZ - Skip next single-word instruction if E is zero.
- SLA - Skip if least significant bit of A is zero.
- SLB - Skip if least significant bit of B is zero.
- SSA - Skip if A is positive.
- SSB - Skip if B is positive.
- SZA - Skip if contents of A equals zero.
- SZB - Skip if contents of B equals zero.

Operands within the alter-skip group can be combined as follows:

$$\left\{ \begin{array}{l} \text{CLA} \\ [\text{CMA}] \\ \text{CCA} \end{array} \right\} [,\text{SEZ}] \left\{ \begin{array}{l} ,\text{CLE} \\ [,\text{CME}] \\ ,\text{CCE} \end{array} \right\} [,\text{SSA}] [,\text{SLA}] [,\text{INA}] [,\text{SZA}] [,\text{RSS}]$$

$$\left\{ \begin{array}{l} \text{CLB} \\ [\text{CMB}] \\ \text{CCB} \end{array} \right\} [,\text{SEZ}] \left\{ \begin{array}{l} ,\text{CLE} \\ [,\text{CMA}] \\ ,\text{CCE} \end{array} \right\} [,\text{SSB}] [,\text{SLB}] [,\text{INB}] [,\text{SZB}] [,\text{RSS}]$$

At least one and up to eight of the alter-skip instructions are included in one statement. Instructions referring to the A-Register cannot be combined in the same statement with those referring to the B-Register. When two or more skip opcodes are combined in a single operation, a skip occurs if any one of the conditions exists. If a statement with RSS also includes both SSA and SLA (or SSB and SLB), a skip occurs only when sign and least significant bit are both set (1).

## Index Register Group

The index register group contains 32 instructions that perform various operations involving the use of index registers, X and Y. Instructions in this group generate calls to external subroutines when used on the L-Series. Statements containing opcodes from this group can take on one of four syntactical forms.

One form of statement using index register opcodes is:

```
[label] opcode [;comments]
```

Opcodes that require this form are:

CAX - copy A to X

CAY - copy A to Y

CBX - copy B to X

CBY - copy B to Y

CXA - Copy X to A.

CXB - Copy X to B.

CYA - Copy Y to A.

CYB - Copy Y to B.

DSX - Decrement X, skip next instruction if result is 0.

DSY - Decrement Y, skip next instruction if result is 0.

ISX - Increment X, skip next instruction if result is 0.

ISY - Increment Y, skip next instruction if result is 0.

XAX - Exchange A and X.

XAY - Exchange A and Y.

XBX - Contents of B and X are exchanged.

XBY - Contents of B and Y are exchanged.

## Machine Instructions

A second form is:

$$[\text{label}] \text{ opcode } \left\{ \begin{array}{l} m \\ @m \\ \text{literal} \end{array} \right\} [;\text{comments}]$$

Opcodes that require this form are:

ADX - Add value of operand to X.

ADY - Add value of operand to Y.

LDX - Load X with value of literal or contents of address specified by operand.

LDY - Load Y with value of literal or contents of address specified by operand.

$$[\text{label}] \text{ opcode } \left\{ \begin{array}{l} m \\ @m \end{array} \right\} [;\text{comments}]$$

Opcodes that require this form are:

JLY - Jump and load Y.

LAX - Load A from memory indexed by X.

LBX - Load B from memory indexed by X.

LAY - Load A from memory indexed by Y.

LBY - Load B from memory indexed by Y.

SAX - Store A into memory indexed by X.

SBX - Store B into memory indexed by X.

SAY - Store A into memory indexed by Y.

SBY - Store B into memory indexed by Y.

STX - Store X into address specified by operand.

STY - Store Y into address specified by operand.

The last statement form using index register opcodes is:

$$[\text{label}] \text{ opcode } \{m\} [;\text{comments}]$$

The opcode using this form is:

JPY Jump indexed by Y.

## No-Operation Instruction

When a no-operation instruction is encountered in a program, no action takes place; the computer goes on to the next instruction. A full memory cycle is used in executing a no-operation instruction.

This instruction can be used in conjunction with the ISZ instruction to perform an increment operation.

General Form:

```
[label] NOP [;comments]
```

## Extended Arithmetic Group (EAG)

The instructions in this group perform extended arithmetic operations on double-word values.

Statements containing opcodes from this group have one of four syntactical forms.

The first form is:

```
[label] opcode  $\left\{ \begin{array}{l} m \\ @m \\ literal \end{array} \right\} [;comments]$ 
```

Opcodes that require this form are:

DIV - Divide the contents of B and A by the value of the literal, or by the contents of the address specified by the operand. The quotient is stored in A and the remainder is stored in B.

DLD - Load the contents of the location specified by the operand and the contents of the following location into A and B, respectively.

MPY - Multiply the contents of A by the value of the literal or by the contents of the address specified by the operand.

## Machine Instructions

The second form is:

$$[\text{label}] \text{ DST } \left\{ \begin{array}{l} m \\ @m \end{array} \right\} [;\text{comments}]$$

This instruction stores the contents of A and B into the address specified by the operand and the following address.

MPY, DIV, DLD and DST result in two machine words, one word for the opcode and one for the operand.

The third form is:

$$[\text{label}] \text{ opcode } n [;\text{comments}]$$

Opcodes that require this form are:

ASL - Arithmetically shift A and B left n bits. The sign bit (bit 15 of B) is unaltered. Least significant bits are zeroed.

ASR - Arithmetically shift A and B right n bits. The sign bit (bit 15 of B) is extended.

LSR - Logically shift A and B right n bits. Most significant bits are zeroed.

LSL - Logically shift A and B left n bits. Least significant bits are zeroed.

RRL - Rotate A and B left n bits.

RRR - Rotate A and B right n bits.

The range of n is from 1 to 16 bits.

The last form of the Extended Arithmetic Group is:

$$\text{SWP } [;\text{comments}]$$

This instruction exchanges the contents of the A and B.



## Input/Output, Overflow, and Halt

The input/output instructions allow you to transfer data to and from an external device via a buffer, to enable or disable external interrupts and to check the status of I/O devices and operations. A subset of these instructions permits checking for an arithmetic overflow condition.

Unlike memory reference instructions, I/O instructions cannot use indirect links.

Input/output instructions require the designation of a select code, *sc*, which indicates one of 64 input/output channels or functions. Expressions used to represent select codes (channel numbers) must have a value of less than 64.

The select code can be a label previously defined as an external symbol by an EXT pseudo instruction. In such a case, the entry point referred to by the pseudo instruction must be an absolute value less than 64. Any other value will be flagged as an error.

Instructions that transfer data between the A- or B-Register and a buffer access the switch register when the select code is 1. The character C appended to such an instruction clears the overflow bit after the transfer from the switch register is complete. For all other select codes C will clear the flag bit on the device.

For example:

LIAC 24

will perform the same action as:

LIA 24

but will also clear the flag bit on select code 24.

The character C can be appended to the following opcodes:

CLC	MIA	OTB	SOC
LIA	MIB	CLO	SOS
LIB	OTA	STC	HLT

Non-privileged programs can only use *sc*=1 (switch register).

Statements containing opcodes from this group can take on one of three syntactical forms.

## Machine Instructions

The first form is:

```
[label] opcode sc [;comments]
```

The Opcodes that require this form are:

- CLC - Clear the I/O control bit for the channel specified by sc. If sc = 0, the control bits for all channels are cleared to zero; all devices are disconnected. If sc = 1, this statement is treated as a NOP.
- CLF - Clear the flag bit to zero for the channel indicated by sc. If sc = 0, the interrupt system is disabled. If sc = 1, the overflow bit is cleared to zero.
- LIA - Load the contents of the I/O buffer indicated by sc into A.
- LIB - Load the contents of the I/O buffer indicated by sc into B.
- MIA - Merge (inclusive "or") the contents of the I/O buffer indicated by sc into A.
- MIB - Merge (inclusive "or") the contents of the I/O buffer indicated by sc into B.
- OTA - Output the contents of A to the I/O buffer indicated by sc.
- OTB - Output the contents of B to the I/O buffer indicated by sc.
- SFC - Skip the next single-word instruction if the flag bit for channel sc is clear. If sc = 1, the overflow bit is tested. If sc = 0, the status of the interrupt system is tested.
- SFS - Skip the next single-word instruction if the flag bit for channel sc is set. If sc = 1, the overflow is tested. If sc = 0, the status of the interrupt system is tested.
- STC - Set I/O control bit for channel specified by sc. STC transfers or enables transfer of data from an input device to the buffer or to an output device from the buffer. If sc = 1 the statement is treated as a NOP.
- STF - Set the flag bit of the channel indicated by sc. If sc = 0 the interrupt system is enabled. If sc = 1, the overflow bit is set.

## Machine Instructions

The second form is:

```
[label] opcode [;comments]
```

Opcodes that require this form are:

CLO - Clear the overflow bit.

STO - Set overflow bit.

SOC - Skip the next single-word instruction if the overflow bit is clear.

SOS - Skip the next single-word instruction if the overflow bit is set.

The last statement form of this group is:

```
[label] HLT [sc[;comments]]
```

or

```
[label] HLTC [sc] [;comments]
```

This instruction halts the computer in privileged mode. If not privileged mode, the instruction generates a memory protect.

If you use neither the select code nor the C option, you cannot use the comments portion of the instruction.



## Floating Point

The instructions in this group perform arithmetic operations on floating-point operands. These instructions make calls to arithmetic subroutines. The operand field can contain any relocatable expression or absolute expression resulting in a value of less than 2000 octal.

The statements containing these opcodes can take one of the following two syntactical forms.

The first format is:

```
[label] opcode      m
                   @m   [;comments]
                   =Fn
```

Instructions that use this form are:

**FAD** - Add the two-word floating-point quantity in A and B to the two-word floating-point quantity in the address specified by the operand and its following location or to the quantity defined by the literal. The result is stored in A and B.

**FDV** - Divide the two-word floating-point quantity in A and B by the two-word floating-point quantity in the address specified by the operand and its following location or by the quantity defined by the literal. The result is stored in A and B.

**FMP** - Multiply the two-word floating-point quantity in A and B by the two-word floating-point quantity in the address specified by the operand and its following location or by the quantity defined by the literal. The result is stored in A and B.

**FSB** - Subtract the two-word floating-point quantity in A and B from the two-word floating-point quantity in the address specified by the operand and its following location or by the quantity defined by the literal. The result is stored in A and B.

## Machine Instructions

The other form is:

```
[label] opcode  [;comments]
```

Instructions that use this form are:

**FIX** - Convert the floating-point number contained in A and B to an INTEGER. The result is returned in A. After execution, the contents of B are meaningless.

**FLT** - Convert the INTEGER in A to a floating-point number. The result is returned in A and B.

## Dynamic Mapping System Instructions

If the computer on which the object program is to be run includes a Dynamic Mapping System, you can use any of the following group of instructions. These instructions may not be legal on your HP 1000. Consult your hardware manual.

Statements containing opcodes from this group can take on one of the following three syntactical forms.

The first form is:

```
[label] opcode  m          m  
                @m        @m    [;comments]  
                literal
```

The instruction that requires this form is:

**JRS** - Jump to the location specified by the second operand and restore status. The first operand contains the address of the status word in memory (or the value of the status word if the operand is a literal).

Operands are separated by a space.

## Machine Instructions

Another form is:

```
[label] opcode [;comments]
```

Opcodes that require this form are:

LFA - Load the contents of the A-Register into the base page fence register.

LFB - Load the contents of the B-Register into the base page fence register.

MBF - Move bytes using the alternate program map for source reads and the current enabled map for destination writes.

MBI - Move bytes using the currently enabled map for source reads and the alternate program map for destination writes.

MBW - Move bytes with both the source and destination addresses established through the alternate program map.

For MBF, MBI and MBW, the A-Register contains the source-byte address and the B-Register contains the destination-byte address. Both addresses must be even numbers. The X-Register contains the number of bytes to be moved.

MWF - Move words using the alternate program map for source reads and the currently enabled map for destination writes.

MWI - Move words using the currently enabled map for source reads and the alternate program map for destination writes.

MWW - Move words with both the source and destination addresses established through the alternate program map.

For MWF, MWI and MWW, the A-Register contains the source address and the B-Register contains the destination address. The X-Register contains the number of bytes to be moved.

PAA - Transfer the 32 Port-A map registers to or from memory according to the address in the A-Register.

PAB - Transfer the 32 Port-A map registers to or from memory according to the address in the B-Register.

## Machine Instructions

- PBS - Transfer the 32 Port-B map registers to or from memory according to the address in the A-Register.
- PBB - Transfer the 32 Port-B map registers to or from memory according to the address in the B-Register.
- RSA - Read the contents of the MEM status register into the A-Register.
- RSB - Read the contents of the MEM status register into the B-Register.
- RVA - Read the contents of the MEM violation register into the A-Register.
- RVB - Read the contents of the MEM violation register into the B-Register.
- SYA - Transfer contents of the system map registers to or from memory, using the A-Register.
- SYB - Transfer contents of the system map registers to or from memory, using the B-Register.
- USA - Load or store the user map according to the contents of the A-Register.
- USB - Load or store the user map according to the contents of the B-Register.
- XMA - Transfer a copy of the system or user map into the Port-A or Port-B map as determined by the control word in the A-Register.
- XMB - Transfer a copy of the system or user map into the Port-A or Port-B map as determined by the control word in the B-Register.
- XMM - Transfer a number of words from sequential memory locations to sequential map registers, or from map to memory.
- XMS - Transfer a number of words to sequential map registers.

## Machine Instructions

The last form is:

[label] opcode  $\left\{ \begin{array}{c} m \\ @m \end{array} \right\}$  [;comments]

Opcodes that require this form are:

- DJP - Disable MEM and jump.
- DJS - Disable MEM and jump to subroutine.
- SJP - Translate all programmed memory references using system map.
- SJS - Translate all programmed memory references using system map.
- SSM - Store contents of the MEM status register into the addressed memory location.
- UJP - Specifies that the MEM hardware will use the user map for translating all programmed memory references. Indirect references are resolved in the current map before accessing the alternate map.
- UJS - Enable user map and jump to subroutine.
- XCA - Compare the contents of the A-Register with the contents of the addressed memory location in the alternate map. Skip next word if contents are unequal.
- XCB - Compare the contents of the B-Register with the contents of the addressed memory location in the alternate map. Skip next word if contents are unequal.
- XLA - Load the contents of the specified memory location in the alternate map into the A-Register.
- XLB - Load the contents of the specified memory location in the alternate map into the B-Register.
- XSA - Store the contents of the A-Register into the addressed memory location in the alternate map. The previous contents of the memory cell are lost; the A-Register contents are not altered.
- XSB - Store the contents of the B-Register into the addressed memory location in the alternate map. The previous contents of the memory cell are lost; the B-Register contents are not altered.

## HP 1000 Fence Registers

There are two separate fences available on the HP1000 M-, E-, and F-Series Computers: the memory protect fence and the base page fence.

The memory protect fence allows you to select a block of memory that will be protected against alteration by any programmed instruction. The memory protect fence register (which specifies the upper bound of the protected area) is loaded from the A- or B-Register by an OTA or OTB instruction. See the appropriate Operating and Reference manual for specific details on the memory protect fence.

The base page fence is only available in HP1000 M-, E- and F-Series Computers, which have the Dynamic Mapping System. This fence specifies which part of the base page is mapped. This determines where shared memory is separated from reserved memory on the base page. The base page fence register is loaded from the A- or B-Register by an LFA or LFB instruction.

Instructions that modify the fence registers cannot be executed while the computer is in the protected mode.

## HP 1000 M-, E-, F-Series Instruction Replacements

The RTE-XL system library contains software substitutions for all the HP 1000 M-, E- and F-Series CPU instructions that are not included in the HP 1000 L-Series instruction set except for the optional DMS instruction set, which is not simulated on the HP 1000 L-Series hardware.

These instruction replacements should enable most user programs written in assembly language to be transported from the HP 1000 M-, E- and F-Series Computers to an HP 1000 L-Series by simply editing those instruction mnemonic codes into the JSB <mnemonic> format for the system library routines in RTE-XL.

If you should happen to code an instruction that is not valid for the L-Series hardware, the Assembler will not report this fact as an error. The Assembler assembles the full HP 1000 instruction set. The L-Series instruction set is a subset of the HP 1000 instruction set. (See Appendix C for a summary of the valid instruction sets for the M-, E-, F- and L-Series Computers.) Since neither the Assembler nor the Loader will report unimplemented instructions, the L-Series processor will trap and report as an error any instruction that is not valid for its instruction set. The following error will be reported on the system console:

```
name ABORTED UI Address
```

where:

```
name      is the name of the program.
```

```
address   is the address of the offending instruction.
```

## Replacement Formats

The name of the software subroutine is formed by preceding the instruction mnemonic with a period (decimal point). The calling sequence is transformed as shown in Figure 3-2. All instructions that are recoded to use the software implementation will be declared as external to the program by the Assembler.

1-word instructions:

```
LABEL XYZ COMMENTS is edited to -->  
LABEL JSB .XYZ COMMENTS
```

2-word instructions:

```
LABEL XYZ <operand> COMMENTS is edited to -->  
LABEL JSB .XYZ COMMENTS  
DEF <operand>
```

3-word instructions (CBT, CMW, MBT, MVW):

```
LABEL MBT <operand> COMMENTS is edited to -->  
LABEL JSB .MBT COMMENTS  
DEF <operand>  
DEC 0
```

3-word instructions (CBS, SBS, TBS):

```
LABEL CBS <operand 1> <operand 2> COMMENTS  
is edited to -->  
LABEL JSB .CBS COMMENTS  
DEF <operand 1>  
DEF <operand 2>
```

Figure 3-2. HP 1000 Replacement Formats





## Replacement Formats

The name of the software subroutine is formed by preceding the instruction mnemonic with a period (decimal point). The calling sequence is transformed as shown in Figure 3-2. All instructions that are recoded to use the software implementation will be declared as external to the program by the Assembler.

<p>1-word instructions:</p> <p>LABEL XYZ COMMENTS is edited to --&gt;</p> <p>LABEL JSB .XYZ COMMENTS</p>
<p>2-word instructions:</p> <p>LABEL XYZ &lt;operand&gt; COMMENTS is edited to --&gt;</p> <p>LABEL JSB .XYZ COMMENTS DEF &lt;operand&gt;</p>
<p>3-word instructions (CBT, CMW, MBT, MVW):</p> <p>LABEL MBT &lt;operand&gt; COMMENTS is edited to --&gt;</p> <p>LABEL JSB .MBT COMMENTS DEF &lt;operand&gt; DEC 0</p>
<p>3-word instructions (CBS, SBS, TBS):</p> <p>LABEL CBS &lt;operand 1&gt; &lt;operand 2&gt; COMMENTS is edited to --&gt;</p> <p>LABEL JSB .CBS COMMENTS DEF &lt;operand 1&gt; DEF &lt;operand 2&gt;</p>

Figure 3-2. HP 1000 M-, E- and F-Series Replacement Formats

## HP 1000 M-, E- and F-Series Software Replacements

The following list represents the HP 1000 M-, E- and F-Series instructions that have software substitutions in the RTE-XL system library.

ONE WORD		TWO WORD	THREE WORD (2 operand) (1 operand)	
.CAX	.FIX	.ADX	.LBY	.CBS
.CAY	.FLT	.ADY	.LDX	.SBS
.CBX		.LDY		.TBS
.CBY	.ISX	.FAD		.MBT
.CXA	.ISY	.FDV	.SAX	.MVW
.CXB		.FMP	.SAY	
.CYA	.LBT	.FSB	.SBX	
.CYB			.SBY	
	.SBT	.JLY	.STX	
.DSX	.SFB	.JPY	.STY	
.DSY				
	.XAX	.LAX		
	.XAY	.LAY		
	.XBX	.LBX		
	.XBY	.LBY		
.SFB		.XSA		
.MBF		.XSB		
.MWF		.JLB		
		.JLA	.XLA	
		.XCA	.XLB	
		.XCB		
		.DIV		
		.DLD	.DST	
		.MPY		

# Chapter 4

## Assembler Instructions

This chapter describes Assembler instructions, also known as pseudo operations or simply pseudo ops. The term "pseudo op" means that these operations are not really machine instructions but instructions used to control the assembly process. For example, they indicate to the Assembler where the program starts or how many words to reserve for an array. Assembler instructions perform the following functions:

**Assembler Control:** Specifies the start and end of a program, assigns blocks of code or data to a memory space, and determines how to include source files in the pending file.

**Loader and Generator Control:** Passes commands to the Loader or the Generator.

**Program Linkage:** Enables communication among subroutines or between a main program and its subroutines.

**List Control:** Determines the list output format.

**Storage Allocation:** Reserves memory for data or for work area.

**Constant Definition:** Defines constants and controls placement of literal values.

**Address and Symbol Definition:** Defines and generates 16 and 32 bit addresses and equates values with symbols.

**Declaring Assembly-Time Variables:** Declares or alters assembly-time variables.

**Conditional Assembly:** Allows assembly on only specified sections of code or repeatedly assembles a set of instructions, takes advantage of declaring user-defined errors.

## Assembler Control

The Assembler control instructions establish and alter the location counters of memory spaces. Before discussing these instructions, the memory spaces and their contents are explained below.

A memory space is an area in computer memory designated by the user to hold executable code or data depending on the application. Each memory space has its own counter that is maintained in the same way as is the program location counter.

The five memory spaces are:

- program relocatable
- base page relocatable
- EMA relocatable
- SAVE relocatable
- common relocatable

The five spaces are shown below in a view of the user's logical memory map.

## Assembler Instructions

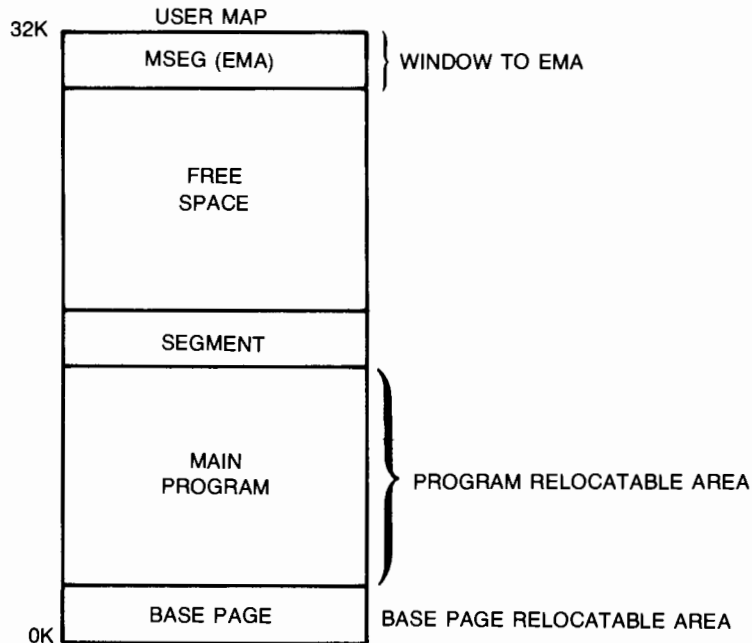


Figure 4-1. A View of the User's Map in Logical Memory

The user program resides in the program relocatable space.

Base page relocatable space is in the base page.

The EMA relocatable space holds data reserved for use by EMA. An MSEG space is a place to hold a small section (two or more pages) of a large EMA array. It can be thought of as a "window" into an EMA array because the MSEG section can be manipulated to point to all of the array in physical memory. For more information on EMA programming, refer to the Programmer's Reference Manual.

The SAVE relocatable space holds variables in much the same way as common holds data. The difference is that variables in common space are free to be changed by other subroutines and segments. Only the subroutine or segment that placed a variable in SAVE space can access it. Even if the segment is overlaid by another segment, the variables in SAVE space are not changed. SAVE spaces may be linked to more than one module if the spaces are allocated with the ALLOC instruction. You must declare how many words of SAVE to reserve at load time.

The common relocatable space holds variables declared to be common.

The Assembler control instructions discussed in this section are:

```
NAM
ORG
RELOC
ORR
END
```

## **NAM**

```
NAM name[,type[,priority[,resolution[,multiple[,hours
      [,minutes[,seconds[,milliseconds]]]]]]]]
      [loader comment]
```

The NAM statement designates the start of a relocatable program. It contains optional parameters defining attributes of the program. These parameters are passed to the loader (see Appendix H for the format of the NAM record).

## Assembler Instructions

The parameters of the operand are optional except for the name, but must be used in the order listed, and must be separated by commas. Also, to specify any particular parameter those preceding it must also be specified.

name - name of program, up to five characters, may be any legal label.

type - program type 1-8, 13, 14, 15, 30, 512 (See Table 4-1).

priority - program priority number (1 to 32767, default = 99).

resolution - resolution code; specifies units to be used with the multiple parameter.

1 = 10's of milliseconds

2 = seconds

3 = minutes

4 = hours



multiple - execution multiple integer (0-4095) that specifies the time interval between runs for programs that run repeatedly. To be used with the resolution parameter. A zero value indicates the program is to be run at once.

hours,  
minutes,  
seconds,  
milliseconds

A typical NAM statement will look like this:

```
NAM XYZ,3
```

Example: This is one way of time-scheduling a program. The resolution code is 3 (time unit is minutes) and the multiplier is 10. This means the program will run every 10 minutes. The program is declared to be type 2 (real time disc resident) with priority 50.

```
NAM repet,2,50,3,10 Runs every 10 minutes
```



## Assembler Instructions

Table 4-1. Program Types

TYPE	MEANING
0	System program or driver.
1	Memory-resident.
2	Real-time disc-resident.
3	Background disc-resident.
4	Background disc-resident without Table Area II access.
5	Program segment (RT or BG).
6	Library, re-entrant or privileged subroutines. If called by a memory-resident program, these routines are relocated into the Memory-Resident Library. After memory-resident loading they become Type 7.
7	Library, utility subroutines (appended to calling program).
8	If program is a main, it is deleted from the system, or if program is a subroutine, it is used to satisfy any external references during generation; however, it is not loaded in the relocatable library area of the disc.
13	System entry points that contain pointers and system values defined at generation. Table Area II is a combination of these relocated type-13 modules and system tables built by the generator.
14	Same as type 6, but automatically included in the Memory-Resident Library. They become type 7 after memory-resident loading.
15	System entry points that must be in the system and user maps. Table Area I is a combination of these relocated type-15 modules and I/O tables built by the generator.
30	Subsystem Global Area (SSGA)
512	Block Data subprogram: a module containing data only.
NOTE:	In some cases the primary type code (i.e., 1,2,3,4) may be expanded by adding 8, 16, 24, or 128 to the number. Refer to the Programmer's Reference Manual for details on program type.

## Assembler Instructions

The comment field can begin after any parameter. A blank within the parameter field will terminate the field and cause Macro/1000 to recognize the next entry as the comment field. Macro will report an error for a comment field that extends beyond column 128. It is placed in the NAM relocatable record and is kept with the NAM record through the loading process. String substitution is performed on the loader comment field. Any assembly-time variables after a semicolon or surrounded by single quotes will not be evaluated.

For example, the following declares a program names PROG to be type 3 (background disc-resident) with priority 99. The space terminates the parameters field and begins the comment field. The source statement is the NAM statement input to the assembler and the loader listing is the output from the LOADR.

source statement:

```
NAM PROG,3,99           Program that does many things
```

loader listing:

```
:RU,LOADR,,%PROG
PROG   40042 40047       Program that does many things
```

You can have assembly-time variables within the comments field. Two system assembly-time variables are specifically set aside for use in the NAM statement, &.DATE and &.DTIME. They cause the system date and time to be printed in the following format:

```
&.DATE - yymodd
&.DTIME - yymodd.hh:mm
```

where:

```
yy   is the year
mo   is the month
dd   is the day
hh   is the hour
mm   is the minutes.
```

## Assembler Instructions

Everytime a module using one of these ATV's is assembled, it is time stamped.

The following NAM statement has a "time stamp" on it, using the system date (&.DATE) assembly-time variable. It was last updated on August 13, 1980.

source statement:

```
NAM suprt Support routine #5317. Date &.DATE
```

listing after assembly:

```
NAM suprt Support routine #5317. Date 800813
```

## Assembler Instructions

### ORG

ORG <operand> [;comment]

The ORG statement:

Defines the origin of an absolute program, or

Defines the origin of subsequent sections of an absolute or relocatable program.

An absolute program must begin with an ORG statement. ORG statements may be used elsewhere in the program to define starting addresses for portions of code. All instructions following an ORG statement are assembled at consecutive addresses starting with the value of the expression in the operand field.

The operand field specifies the initial setting of the program location counter. It may contain:

an integer

an absolute symbol previously defined

an assembly-time variable previously defined

an expression

If the ORG statement appears within an absolute program, any type of expression is legal; if the ORG statement appears within a relocatable program, common relocatable expressions must not be used, because the loader will not accept those records.

The ORG statement must not contain an external reference.

## Assembler Instructions

Example:

```

        NAM    MAIN
INIT    BSS    1           ;Initialization section.
        :
        JMP    @INIT
        ORG    INIT       ;Set a relocatable origin at INIT.
DATA    BSS    50         ;Reserve room here for data.
        ORR
;
MAIN    NOP              ;Main program starts here.
        JSB    INIT       ;Go to initialization section.
        LDA    FOO        ;We may now overlay the initialization
        STA    DATA      ;section.
        STB    DATA+1
        :
        END    MAIN
```

In the preceding example, the block of code starting with INIT serves two purposes. It is an initialization routine for this program, and it is also an area to hold data. ORG INIT sets the origin of a relocatable space at relocatable location INIT.

Example:

```

        NAM    BUFR
        ENT    BUFR
Buff.size EQU 128       ;Declare the buffer size and
BUFR     BSS    Buff.size ;reserve that many words for it.
        ORG    BUFR       ;Set the absolute base at top of buffer.
        ABS    Buff.size  ;The first word of the buffer is the size.
        ORR
        END
```

The routine BUFR creates an area of Buff.size (128) words. The first word in the buffer is its length.

## RELOC

RELOC keyword [;comment]

The RELOC statement allows you to designate to which relocatable space the following code or data is to be assembled.

The operand can be one of the following five keywords. Only the first three characters of the keyword are required:

- PROG Assembles the ensuing code or data into the program relocatable space.
- COMMON Assembles the ensuing code or data into the unnamed common space.
- BASE Assembles the ensuing code or data onto the base page.
- SAVE Assembles the ensuing code or data into the SAVE relocatable space.
- EMA Reserves the space that follows RELOC instruction in EMA. This is a special case of the RELOC command. Only BSS commands are supported in EMA space; any other statement will have unpredictable results. Any labels in this space are local names used to refer to local EMA via DDEF statements since they represent 32-bit values.

If no RELOC space is specified, the assembler assumes the program relocatable space.

The scope of the RELOC statement is terminated when another RELOC statement is used.

See the introduction to assembler control pseudo ops for more information about memory spaces.

Within the EMA and COMMON relocation spaces, no code generation or initialization is permitted:

```
NAM FOO
:
RELOC COMMON
LDA A ; illegal. Generates code.
DEF * ; illegal.
ABS 2 ; illegal.
:
```

This error may be detected at load time.

## Assembler Instructions

Example using RELOC statement:

```
        NAM    MAIN
        :
        RELOC  COMMON
X       BSS    10      ; Reserve words in common.
Y       BSS    10
        RELOC  SAVE
FLAG    DEC    1      ; This goes in the SAVE space.
;
        RELOC  PROG
MAIN    NOP          ; The program starts here.
        :
        END    MAIN
```

## ORR

```
ORR [;comment]
```

The ORR statement terminates the absolute or relocatable mode set by an ORG statement.

This statement has no label and no operand.

More than one ORG statement may occur before an ORR is used. If so, when the ORR is encountered, the memory space specified before the first ORG statement will take precedence.

If more than one ORR appears after an intervening ORG, an error occurs.

Example:

```
        RELOC  PROG
FIRST   NOP
        :
        ORG    FIRST +2500 ;Set an origin in Relocatable space.
        :
        ORG    FIRST +2900 ;Set another origin in Relocatable space.
        :
        ORR                    ;Back to program relocatable space.
```

### **END**

END [name][;comment]

The END statement terminates the program module. It marks the physical end of the set of source language statements whose name is indicated on the preceding NAM statement. It does not, however, mark the end of the source input.

The operand field contains a name appearing as a statement label in the current program; or, it may be blank. If the pending module is the main program, the operand field must be specified. The name identifies the transfer address where RTE is to begin execution.

### **Multiple Modules**

Any number of modules (defined as a set of assembly statements starting with a NAM and ending with an END) may be concatenated together in one file and assembled together. Only one control statement is allowed per file. When a NAM statement is encountered, Macro clears its symbol tables and prepares to assemble this new module.

Macro definitions, assembly time-variable values, conditional assembly, and MACLIB declarations exist for the entire file, while user labels, RELOC declarations and local data are recognized only within the module in which they are defined. Only conditional assembly statements, Macro definitions, assembly-time variable declaration statements, comments, blank lines or a NAM statement may follow an END statement.

Absolute programs must be submitted in one module.



## Assembler Instructions

Example:

```
MACRO,L,R
  MACRO
    TYPE &MSG          ; macro to type a message to the terminal.
;
; Macro definitions can come before NAM statements and must be
; defined before they are used.
;
  EXT EXEC
  JSB EXEC
  DEF *+5
  DEF =D2              ; EXEC 2, output.
  DEF =D1              ; To LU 1.
  DEF =S&MSG          ; Define the message.
  DEF =L-:L:&MSG      ; Pass the negative # of characters.
ENDMAC
;
;
  MACRO
    QUIT              ; Macro to call exit.
    EXT EXEC
    JSB EXEC
    DEF *+2
    DEF =D6
ENDMAC
;
; First module
;
  NAM first
  EXT second
first  NOP            ; Entry point of first module.
      TYPE 'Hi there #1' ; Call the Macro defined above.
      :
      JSB second
      END first      ; 'first' is the transfer address, it defines
                    ; the executable start of this module.
;
; Second module
;
  NAM second
  ENT second
second NOP           ; Entry point of second module.
      TYPE 'Hi there #2' ; Call the same Macro defined above.
      QUIT
      END
```

## INCLUDE

```
INCLUDE namr
```

The INCLUDE pseudo op causes the assembler to continue assembly from the file specified in the operand field.

The operand field contains the RTE file name of the file to be included. The file name can be represented by a single assembly time variable, macro parameter, or can be explicitly entered. The operand will be folded to upper case.

To include a file whose name starts with an ampersand (&), the file name must be surrounded by single quotes:

```
INCLUDE '&FILE::CR'
```

The included file must consist of legal source code and may contain INCLUDE pseudo ops. These are referred to as nested include files. Only five levels of nesting are allowed.

When the assembler encounters the INCLUDE statement, it begins the line numbering for the listing at line number one and appends the letter I to the line number (the number of the current source line is saved). A file number is appended to the page number in the listing. This is done so that any errors found in this file will reflect the actual line number of the include file for easy correction.

When the end of the included file is reached, assembly continues at the statement following the INCLUDE statement in the file where it appeared. The line numbers resume from the INCLUDE statement.

The comment field is not allowed.

## Assembler Instructions

Example:

```
        NAM TEST
        INCLUDE DATA::CR
;
; Include a file here that contains data initialization,
; storage areas, and common declarations.
;
        :
TEST   NOP
        :
        END TEST
```

This is what will be assembled:

```
        NAM TEST
;
; from file DATA
;
        RELOC COMMON
ABC    DEC 10,20,30,40,50
ARRAM  BSS 50
        RELOC PROG
LU     DEC 1
        :
TEST   NOP
        :
        END TEST
```

## Loader and Generator Control

This section covers two special pseudo ops, LOD and GEN. They are not instructions to the assembler. They are a means to pass commands from the assembler to certain loaders or generators. Consult your Programmers Reference Manual or Loader Manual for further detailed information.

### LOD

```
LOD n,string [;comment]
```

The LOD statement is an instruction to the loader.

When the loader encounters a LOD statement, it performs the function defined by the operand field.

The operand has two parts:

n - states the number of words in the character string.  
 string - any legal loader command allowed before the SEarch or RElocate commands.

Example:

```
MACRO,L,R
  NAM LOAD
  LOD 3,OP,DB          ;tell the LOADR to append DBUGR to this
                      ;program.
  LOD 3,SZ,32         ;size this program to 32 pages.
LOAD  NOP             ;program starts here.
  :
  :
  END LOAD
```

## GEN

```
GEN n,string [;command]
```

The GEN statement is an instruction to some generators. The instruction is contained in the string portion of the operand.

The operand contains two portions:

- n - The number of words in the string.
- string - The instruction to the generator.

Example:

```
MACRO,L,R
    NAM drivr
    GEN 14,EDD.00,TX:15,TO:32000
drivr NOP                ;program begins here.
    :
    :
    END drivr
```

## Program Linkage

The linking pseudo instructions provide a means for communication between a main program and its subroutines or among several subroutines to be run as a single program. These instructions may be used only in a relocatable program. The following pseudo ops are discussed in this section:

```

ENT
EXT
ALLOC
RPL

```

### ENT and EXT

```
ENT name=['alias'][,name=['alias']]...[;comment]]
```

```
EXT name=['alias'][,name=['alias']]...[;comment]]
```

The EXT pseudo instruction declares that symbols used in this module are to be linked to an external routine. The symbols must be defined as entry points in another module. They may appear in memory reference instructions, certain I/O instructions, EQU or DEF pseudo ops. An external symbol can be used with a + (plus) or - (minus) offset or specified as indirect.

The ENT pseudo instruction declares entry points that are to be defined in the module. Each name is a symbol, usually a data-type statement or a NOP that is assigned as a label for some statement in the program. Entry points allow another module to refer to this module. All entry points must be defined in the module.

The operand field contains:

name is the name of the entry point (for ENT) or the name of a label external to this program (for EXT). Any number of ENT names and up to 2047 EXT names can be specified per module by the user.

='alias' gives the entry point or external label another name. See the discussion that follows.

You may have more than one EXT statement in a module containing the same symbol. Likewise, you may have more than one ENT statement in a module pointing to the same symbol. Each duplicate ENT or EXT statement is ignored.

## Assembler Instructions

Example:

```

        NAM MAIN
        EXT subroutine    ; Declare 'subroutine' to be external.
MAIN    NOP
        :
        JSB subroutine    ; Jump to subroutine.
        END MAIN

        NAM SUB
        ENT subroutine    ; Declare 'subroutine' to be an entry
subroutine NOP            ; point in this program.
        :
        JMP @subroutine   ; Jump back to main.
        END
```

### Alias

There are some cases in which you may wish to refer to an external routine whose name may not be a legal label in the Macro/1000 language. You may also wish to define an entry point bearing an illegal label for reference by other programs. To do this, he may equate a legal label to an illegal label:

```
        ENT LEGAL='$/OOP'
```

Since `$/OOP` is an illegal label in Macro/1000, it cannot be referenced. External routines can gain entry to this routine by using `$/OOP`; it is the actual entry point. The name for only this module is 'LEGAL'.

```
        ENT LEGAL='$/OOP'
LEGAL  NOP                ; entry point
```

In the same manner, you can use the alias option on external declarations:

Example:

```
        EXT GOOD='#[LAB'
        :
        JSB GOOD                ; Calls the external routine, #[LAB.
```

where:

`#[LAB` is a label in an external routine.

`GOOD` is the symbol that is to be used to reference the routine.

## Assembler Instructions

### ALLOC

```
label  ALLOC keyword,#words [,MSEG size] [;comment]
```

The ALLOC statement pseudo op allocates or sets up a link to a globally accessible named EMA space or a SAVE space.

The label is the name of the EMA, SAVE, or COMMON space.

The keyword can be one of three words:

EMA If the keyword is EMA, the following parameter specifies how many words are to be in this space. The third parameter is optional and is the MSEG size in pages. Macro keeps track of the MSEG sizes and passes the largest size to the loader.

SAVE If the keyword is SAVE, specify how many words are to be in this space.

COMMON If the keyword is COMMON, specify how many words are to be in this space. This common block can be linked to a FORTRAN named common block.

The EMA instruction cannot be used in the same program as the ALLOC EMA or RELOC EMA commands. Appendix L has information on the EMA instruction.

Through the use of the ALLOC command, values in the EMA, SAVE, or the COMMON space can be shared between modules.



## Assembler Instructions

Example:

```

      NAM   MAIN
Q     ALLOC EMA,50000,2    ; Declare Q to be a 50000 word
                          ; array in EMA.
EMA.ADDR DDEF Q          ; This is the only way the
      :                  ; label Q may be used.
GSAV  ALLOC SAVE,100     ; Reserve 100 words of SAVE
      :                  ; space.
      END

;

      NAM   SUBR
Q     ALLOC EMA,50000,2    ; Declare Q to be in EMA.
      .                  ; Same EMA as declared in MAIN.
      .                  ; Shared with MAIN, and any
      .                  ; other module that declares
      .                  ; it. Linked by loader.
GSAV  ALLOC SAVE,100     ; Same SAVE space as in MAIN.
      END

```

Note this difference between ALLOC and RELOC:

ALLOC globally declares SAVE, EMA or COMMON spaces, i.e., modules may be linked to the same space.

RELOC locally declares SAVE, EMA, PROG, etc. spaces, i.e., external modules do not have direct access to the local spaces.

Example:

```

      NAM   MAIN
QEMA  RELOC EMA          ; Use EMA space.
      BSS   20000        ; Reserve 20000 locally accessible
      :                  ; EMA locations.
      END

;

      NAM   SUBR
QEMA  RELOC EMA          ; QEMA in SUBR is a different EMA
      BSS   20000        ; space than QEMA in MAIN.
      :
      END

```

## Assembler Instructions

### RPL

```
label RPL instruction word [,value][;comment]
```

The RPL pseudo op defines a code replacement record for the RTE system generator or RTE relocating loader.

The label is the mnemonic for the instruction to be replaced by microcode.

The operand is the value of the microcoded instruction word. The operand may also contain a second word that makes up a code replacement value of two words.

Examples:

```
.FAD RPL 105000B  
.VSB RPL 101460B,40B ; This is a two word replacement.
```

Wherever the loader or generator encounters a JSB .FAD, the octal representation of the machine instruction (105000B) will be inserted. There are some constraints:

The instruction to be replaced must be a memory reference instruction (i.e., JSB) or a DEF.

The operand of the memory reference instruction or DEF must be an external reference.

The RPL statement must be in a module separate from the memory reference instructions or DEF.

A two-word RPL will cause the referencing instruction and the next word to be replaced with the microcode replacement.

## Assembler Instructions

Example:

```
NAM RPLAC
.FAD RPL 105000B
:
END
```

Next subroutine:

```
NAM SEG
EXT .FAD
:
JSB .FAD
:
END
```

When this program is loaded, instead of JSB .FAD (whose instruction code in octal is 14XXX), an instruction code of 105000B is relocated.

## Assembly Listing Control

The assembly listing control pseudo ops regulate the assembly listing output.

The following pseudo ops are discussed in this section:

COL	SKP	SUP
HED	SPC	UNS
LIST	SUBHEAD	



### COL

COL column#,column#,column# [;comments]

The COL pseudo op allows you to determine in which columns of the listing the opcode, operand and comment fields will appear.

The operand field contains three parameters. The first parameter specifies the starting column of the opcode field in the listing. It must be greater than 1 and must be less than the second parameter.

The second parameter specifies the starting column of the operand field in the listing. It must be less than the third parameter.

The third parameter specifies the starting column of the comment field in the listing.

The parameters can have values from 2 to 128. These column numbers are relative to the starting column of the legal field in the listing.

If any of the statement fields is large enough that it prevents the following field from beginning in the column specified, a blank is inserted between the fields.

This instruction may appear anywhere in the source, but will be overridden by column indicators in a MACRO statement.

Example:

```
COL 23,32,37
```

requests the opcode to be listed in column 23, the operand in column 32 and the comment in column 37.

## Assembler Instructions

### HED

HED heading

The HED pseudo instruction specifies a heading to be printed at the top of each page of the source program listing. This header is printed in addition to the standard header printed by Macro/1000.

The operand field contains a string of up to 56 ASCII characters that will be printed as a heading at the top of each page of the source program listings.

When a HED pseudo op occurs in a program, the heading will be printed on the following page, below the standard heading printed by Macro/1000. The heading will appear at the top of each successive page, until changed by another HED instruction.

Example:

```
MACRO,L,R
  NAM S2317
  HED Support Subroutine #2317
S2317  NOP
      :
      END
```

Causes the second page of the listing to appear:

```
PAGE# 2  Macro/1000  Version 1.0  4:46 PM TUE, 9 DEC 1980
Support Routine #2317

      S2317  NOP
```

## SUBHEAD

### SUBHEAD subheading

The SUBHEAD pseudo op specifies a subheading to be printed on the listing, and creates a table of contents at the end of the listing.

The operand field contains a string of 56 ASCII characters to be used as the subheading. This string will be printed on the page following the one in which the command appears. The subheading will appear on the line following the heading (if heading exists) or immediately below the standard macro heading.

A table of contents is created at the end of the listing and contains all subheads and the pages on which they occur.

Example:

```
MACRO,L,R
      NAM S2317
      HED Support Subroutine #2317
      SUBHEAD integer to real conversion
S2317  NOP
      :
      END
```

Causes the record page of the listing to appear:

```
PAGE#2  Macro/1000  Version 1.0   4:46 PM  TUE,  9 DEC 1980
Support Subroutine #2317
Integer to real conversion
```

```
S2317  NOP
```

At the end of the listing:

```
PAGE#2:  SUBHEAD
Integer to real conversion
```

would appear.

## LIST

LIST keyword [;comment]

The LIST pseudo op alters the current listing state.

The operand field contains a keyword that defines what the new state will be. The keywords which may appear in the operand are:

ON  
OFF  
SHORT  
MEDIUM  
LONG

- ON - This operand instructs Macro/1000 to begin listing the source. If a previous part of the source has been listed, the listing will resume in that state. If no previous part of the source has been listed, the current listing state will be defaulted to 'SHORT'.
- OFF - This operand suppresses the assembly listing, beginning with the 'LIST OFF' pseudo instruction and continuing until the listing is resumed by a 'LIST ON', 'LIST SHORT', 'LIST MEDIUM', or 'LIST LONG'. Any diagnostic messages encountered by Macro/1000 while the listing is off will be printed. The source statement sequence numbers are incremented for instructions skipped.
- SHORT - This operand instructs Macro/1000 to begin an abbreviated listing of the source. No macro definitions or conditional assembly statements appear. No macro expansions appear in the listing, only the Macro call statement. This is the default listing mode. It is best suited for following general program flow.
- MEDIUM - This operand instructs Macro/1000 to begin listing all lines that contain executable statements. No conditional assembly statements appear. Any repeated lines of code are listed. This mode is best used with low-level debuggers since the listing matches the generated code the most.

## Assembler Instructions

LONG - All lines of code that appear in the source and any lines in macro expansions and repeated statements appear. All conditional assembly statements are listed. This mode is best for debugging macros and usages of conditional assembly.

A count is kept of the number of times the listing has been initiated through the use of the 'LIST ON', 'LIST SHORT', 'LIST MEDIUM' or 'LIST LONG' instructions, or 'L' parameter of the control statement. Also, a count is maintained of the number of 'LIST OFF' instructions.

To entirely suppress the listing, the number of 'LIST OFF' instructions must equal the number of times the listing has been initiated. Likewise, to resume listing, the total number of times the listing has been initiated, must be greater than the total number of 'LIST OFF' instructions.

Example:

```
MACRO,R,L
;
; Listing mode is defaulted to short, do not
; list this macro definition:
;
    MACRO
        TEST
        :
    ENDMAC
;
    LIST MEDIUM      ; List only assembled instructions
                    ; in conditional assembly.
;
    AIF &.RS1 = OK
        NAM PROGA
PROGA    NOP
        :
        END PROGA
;
    AENDIF
```



# Assembler Instructions

This is what is listed when run with the following runstring:

```
RU,MACRO,&PROGA,1,-,,,&.RS1='OK'
```

PAGE# 1            Macro/1000                            2:54 PM THU., 26 MAR., 1981

```
00001                    MACRO,L,R
00002                    ; Listing mode is defaulted to short, do not
00003                    ; list this macro definition:
00004                    ;
00009                    ;
00011                                                    ; in conditional assembly.
00012                    ;
00014                                                    NAM PROGA
00015    00000 000000    PROGA    NOP
00016    00001 000000                    :
00017                                                    END PROGA
00018                    ;
Macro:    No errors total
```

## Assembler Instructions

### SKP

SKP [;comment]

The SKP pseudo instruction is a page advance to the list device, The source program listing continues printing at the top of the following page. The SKP instruction is not listed, but the source line sequence number is incremented for each SKP.

Example:

```
ROUT    SKP  
        NOP    ; Start of a routine.
```

The heading (via the HED pseudo op) and the subheading (via the SUBHEAD pseudo op) are printed at the top of the next page.

## SPC

```
SPC n,[m] [;comments]
```

The SPC pseudo op causes a specified number of lines to be skipped in the source program listing.

The operand field can contain two parameters.

The first parameter is an absolute expression, *n*, which specifies the number of lines to be skipped. If the bottom of the page would be reached before *n* lines have been skipped, the output listing will continue printing at the top of the next page.

The second optional parameter contains an absolute expression, *m*, the number of lines which must be left on the bottom of the page after *n* lines have been skipped. If the *m* value is present, and less than *m* lines would be left on the page after *n* lines had been skipped, the output listing will continue printing at the top of the next page.

Example:

```
SPC 5,6
```

means to skip 5 lines and there must be 6 lines left at the bottom of the page.

Do not use the comma unless you use the second parameter:

```
SPC 5, ; error: trailing comma illegal.
```

## Assembler Instructions

### SUP

SUP [;comment]

The SUP pseudo op causes only the first line of code to be listed for those instructions that generate multiple lines. The additional lines generated by the instruction are suppressed until the UNS or the END pseudo op is encountered.

Instructions that generate more than one word of code, and are affected by the SUP pseudo op are:

ADX	DJS	LBX	SBX
ADY	DLD	LBY	SBY
ASC	DST	LDX	SJP
BYT	FAD	LDY	SJS
CBS	FDV	MBT	STX
CBT	FMP	MPY	STY
CMW	FSB	MVW	TBS
DEC	JLY	OCT	UJP
DEY	JPY	SAX	UJS
DEX	LAX	SAY	XMM
DJP	LAY	SBS	XMS

The SUP pseudo op can also be used to suppress the printing of literals at the end of the source program listing and to suppress the printing of offset values for memory reference instructions that refer to external symbols with offsets.

## Assembler Instructions

Example:

```
ASC 4,Hi there
```

will list this code:

```
044151 ASC 4,Hi there
020164
064145
051145
```

while

```
SUP
ASC 4,Hi there
UNS
```

will list:

```
044151 SUP
ASC 4,Hi there
UNS
```

## UNS

```
UNS [;comments]
```

The UNS pseudo instruction causes macro to resume the printing of lines suppressed by a SUP instruction.

## Storage Allocation

The storage allocation pseudo ops reserve a block of memory for data or for a work area.

The pseudo ops covered in this section are:

```
BSS
MSEG
```

### BSS

```
label BSS #words [;comment]
```

The BSS pseudo op reserves up to 32767 words in memory as designated in the operand. The words are reserved in continuous locations in the memory space last declared in the RELOC statement, or in program relocatable space if no RELOC is used.

If the last RELOC space declared were EMA relocatable space, you could specify up to 2147483647 ( $2^{31}-1$ ) words in memory.

The label, if specified, is the name assigned to the storage space and represents the address of the first word.

The operand can be any expression that results in a positive integer. This integer defines how many words are to be reserved for this space. Any symbols used in the operand must have been previously defined.

Example:           Using arrays

```
ARRAY BSS 100 ; Declare an 100-element array.
pointer DEF ARRAY ; Pointer into array.
:
LDA ARRAY+3 ; Load the fourth element of the array.
```

## Assembler Instructions

### MSEG

```
MSEG size [;comment]
```

The MSEG pseudo op declares the MSEG size for EMA references.

The label field is accepted on this statement, but any references to label will point to the statement following label.

The size is an integer from 2 to 32 representing the number of pages of the MSEG. It can be an EQU value. Macro keeps track of the MSEG sizes requested and passes the largest size on to the loader.

Example:

```
MSEG 10          ; Declare MSEG size of 10 pages.  
MSEG 2           ; Declare MSEG size of 2 pages.
```

For this program, the loader would set aside 10 pages for the MSEG space. For more information on EMA programming, refer to the Programmer's Reference Manual.

## Constant Definition

The constant definition pseudo instructions store one or more constant values into consecutive words of the object program. By assigning labels to statements containing these opcodes, other statements can access the constant values.

The following pseudo ops are discussed in this section:

ASC	DEY
BYT	LIT
DEC	LITF
DEX	OCT

### ASC

```
[label] ASC n,string [;comments]
```

The ASC pseudo op enters a string of ASCII characters into consecutive words of the object programs.

The label field can contain a label that represents the address of the first two characters.

The operand field contains two parameters separated by a comma.

The first parameter, *n*, is an expression resulting in an unsigned decimal value. This parameter determines the number of words used to store the ASCII characters.

The second parameter is a group of ASCII characters to be stored. Anything in the operand field following *2n* characters is treated as a comment. If the end of the statement is reached before *2n* characters have been read, the remaining characters are assumed to be blanks and are stored as such. This statement cannot be continued.



## Assembler Instructions

To store code for non-printing ASCII symbols (for example, CR and LF), use the OCT pseudo instruction.

Example 1:

```
ASC 4,'Hi there'
```

Example 2: Use the length attribute (:L:) to find the length of a string.

```
&MSG    CGLOBAL 'Hi there'  
ASC :L:&MSEG+1/2,&MSEG
```

## BYT

```
[label] BYT constant,constant,... [;comments]
```

The BYT pseudo op generates octal constants in consecutive byte locations of memory.

The label field can contain a label representing the word address of the first constant.

The operand field contains one or more octal constants. These constants must consist of one to three octal digits within the range of 0 to 377, and can be signed. If a constant is unsigned, it is assumed to be positive. If the constant is negative, the two's complement of the absolute value is stored. Since the constants are assumed to be octal, the letter B must not be used.

If the operand field contains an odd number of constants, bits 0-7 of the final word generated are clear (zeros).

Examples:

Define an array of octal byte constants:

```
ARRAY BYT 1,4,7,12,15,20
```

## DEC

[label] DEC constant,constant,... [;comments]

The DEC pseudo op enters one or more decimal constants into consecutive words of the object program.

The label field can contain a label representing the address of the first constant.

The operand field must contain one or more decimal constants. The constants can be either integer or floating point, and may be signed. (If no sign is specified, the constant is assumed to be positive.)

### Integer Numbers

An integer number must be in the range of -32768 to 32767 and is stored in one word.

### Floating-Point Numbers

A floating-point number has two components: a fraction, n, and a signed exponent, e. The floating point number must be in the range of:

$$1.469368 \times 10^{-39} \quad \text{to} \quad 1.701412 \times 10^{38}$$

and have one of the following formats:

n.n	n.nEe
n.	n.Ee
	.nEe
	nEe

## DEX

```
[label] DEX constant,constant,... [;comments]
```

The DEX pseudo instruction enters a string of extended precision constants into consecutive words of the object program.

The label field can contain a label representing the address of the first constant.

The operand field must contain one or more decimal constants. The constants can be integer or real but are stored in three consecutive words of memory as extended-precision floating-point numbers.

## DEY

```
[label] DEY constant,constant,... [;comments]
```

The DEY pseudo instruction enters one or more double-precision decimal constants into consecutive words of the object program. It is similar to the DEX pseudo op but stores each constant into four words of memory rather than three.

The label field can contain a label representing the address of the first constant.

The operand field must contain one or more decimal constants. The constants can be integer or real, but are stored in four consecutive words of memory as double-precision floating-point numbers.

## LIT

```
LIT [;comments]
```

The LIT command specifies where the literal block will be placed by Macro/1000. If you do not use this command, MACRO will store the literals at the end of the program.

When the first LIT command is encountered in a program, all literals used in the program up to that point will be stored after it.

Any literals used after the appearance of the LIT command, and not previously defined, will be stored at the end of the program, or following a subsequent LIT command.

### Examples:

```

    LDA =D5
    ADA =D7
    :
    JMP OVER
    LIT
;
; The values of the literals =D5 and =D7 will be stored in the
; next two words.
;
OVER LDA =D5 ; This literal is stored above.
    LDA =D9 ; This literal is stored at the end of the program.
```

## Assembler Instructions

### LITF

```
LITF [;comments]
```

The LITF pseudo op is similar to the LIT pseudo op. It specifies where the literals used in a program will be stored. If the command is not used, the literals will be placed at the end of the program.

However, when LITF command is encountered, all literals defined between the last LITF (or beginning of program) and this LITF will be stored following this LITF command. Even if a memory location has already been assigned to store the value of a literal, that literal value may also appear in another memory location. This implies that all occurrences of a specific value will not necessarily reference the same memory location.

Example:

```
LDA =D5
ADA =D7
:
JMP OVER
LITF
;
; The literals =D5 and =D7 are stored in the next two words.
;
OVER LDA =D5      ; These two literals are stored either at the end
ADA =D7          ; of the program or after the next LITF statement.
```

## Assembler Instructions

### OCT

```
[label] OCT constant,constant,... [;comments]
```

The OCT pseudo op enters one or more octal constants into consecutive words of the object program.

The label field can contain a label representing the address of the first constant.

The operand field contains one or more octal constants. Each octal constant consists of one to 6 octal digits (range of 0 to 177777) and can be signed. If the sign is negative, the two's complement of the absolute value is stored. If the constant is unsigned, the sign is assumed to be positive.

The letter B must not be used after the constant in the operand field: it is significant only when defining an octal term in an instruction other than OCT or BYT.

Example:

Define an array of octal constants:

```
OCT 4,40,400,4000,40000,100000
```

## Address and Symbol Definition

The pseudo operations in this group generate word and byte addresses, or assign a value to a symbol used as an operand elsewhere in the program.

The pseudo ops covered in this section are:

DEF	EQU
DDEF	DBL
ABS	DBR

### DEF

```
label DEF <operand> [;comment]
```

The DEF pseudo op generates one word of memory as a 15-bit address.

The label corresponds to the address at which the DEF resides.

The operand field can be any of the following:

- A relocatable or an absolute expression in a relocatable program.
- An external reference.
- A literal.
- A positive expression in an absolute program.

Operands referring to EMA are not permitted.

The address generated by the DEF pseudo op can be used as the object of an indirect address found elsewhere in the source program. The expression in the operand can itself be indirect and make reference to another DEF statement in the source program.



## Assembler Instructions

Example:

```
        LDA @SYM      ; Load A-Register with the value at the address
        .             ; pointed to by SYM (@ is the indirect address
        .             ; indicator).
SYM DEF NUM          ; The 15-bit address of NUM is stored here.
NUM DEC 10          ; This is the final address, 10 is loaded in
                   ; the A-Register.
```

The operand can be an external routine.

Example:

```
        EXT SQRT      ; SQRT is an external routine.
        JSB @XSQ      ; Get the square root routine.
        :
XSQ DEF SQRT         ; After the indirect is resolved, the 15-bit
                   ; address stored here - the address of SQRT, is
                   ; the object of the JSB.
```

The DEF pseudo op can also be used to hold subroutine parameters following the JSB.

Example:

```
        EXT subroutine
        JSB subroutine
        DEF P1        ; The 15-bit address of P1 and P2 are stored
        DEF P2        ; here.
        :
P1 DEC 10
P2 DEC 11

        ENT subroutine
subroutine NOP        ; The return address is stored here.
        LDA @subroutine ; Load the address of P1.
        STA address    ; Keep it.
        LDA @address   ; Load the value of P1.
```

## DDEF

```
label DDEF <operand> [;comment]
```

The DDEF pseudo op generates a 32-bit (double-word) relocatable address. The first word is the low-order part of a 32-bit EMA relocatable address; the second word is the high-order part.

The label can be used as an operand in a memory reference instruction. It will represent a 15-bit address at which a 32-bit address resides.

The operand field can be:

- A relocatable or an absolute expression in a relocatable program.
- An operand of the EXT pseudo op.
- A label of RELOC EMA or ALLOC EMA block.
- A double-word constant.

The 32-bit address cannot be indirect.

For labels defined using the RELOC EMA or ALLOC EMA statements, the value generated is a true 32-bit address since EMA is being referenced. For any other type of label a word of zeros followed by a 16-bit address results.

## ABS

```
label  ABS  { integer          }  [;comment]
          { absolute expression }
```

The ABS pseudo op defines a 16-bit absolute value.

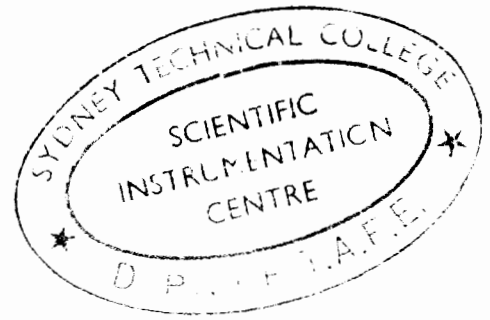
The label is optional, but if specified, represents the value defined by the operand.

The operand can be any absolute expression. Any single symbols of an expression must be defined as absolute elsewhere in the program.

### Examples:

```
AB EQU 35      ; Assigns the value of 35 to the symbol AB
M35 ABS -AB    ; M35 contains -35.
P35 ABS AB     ; P35 contains 35.
P70 ABS AB+AB  ; P70 contains 70.
M36 ABS -36    ; M36 contains -36.
```

## Assembler Instructions



### EQU

```
label EQU <operand> [;comment]
```

The EQU pseudo op assigns to the symbol in the label field the value represented by the operand field.

The label field can be any legal symbol.



The operand field can be any legal expression. The value of the expression can be common, base page, SAVE, external or program relocatable as well as absolute, or any arithmetic combination of these values. The expression can be negative. It cannot be indirect.

The EQU instruction can be used to give a value to a symbol.

Symbols appearing in the operand field must be previously defined in the source program. Duplicate EQU statements will be ignored.

The EQU statement does not result in a machine instruction. Once a label has been equated to a value by use of EQU, its value cannot be changed.

If the user wishes TABLE.A and TABLE.B to occupy contiguous memory locations, he could reserve 5 locations for each table with separate BSS statements. However, to protect against accidentally inserting another value between tables, the following is recommended:

```
TABLE.A BSS 10 ; Defines a 10 word table, TABLE.A.  
TABLE.B EQU TABLE.A+5 ; Equates words 6 through 10 of TABLE.A  
 ; and TABLE.B.  
LDA TABLE.B+1 ; Same as LDA TABLE.A+6
```

## Assembler Instructions

Example:

```
Y EQU *  
X NOP
```

Now X and Y are equivalenced, i.e., they are symbols for the same location.

Two EQU statements are implicit in every Macro/1000 program. They are:

```
A EQU 0  
B EQU 1
```

A and B are reserved symbols representing the A and B-Registers. These symbols cannot be altered.

## DBL and DBR

```
label  DBL  <operand> [;comment]
label  DBR  <operand> [;comment]
```

The DBR and DBL pseudo ops define byte addresses. They each generate one word of memory that contains a 16-bit byte address.

The label is the name of the location containing the byte address. The generated word may be referenced (via label) in the operand field of memory reference instructions elsewhere in the program for the purpose of loading or storing byte addresses.

The operand can be:

- A literal.
- A positive expression in an absolute program.
- An absolute or relocatable expression in a relocatable program.
- A reference to an external.
- A reference to a relocatable space.

Indirect addressing cannot be used.

For DBL, the byte address being defined is the left half (bits 8-15) of the word location declared in the operand.

For DBR, the byte address being defined is the right half (bits 0-7) of the word location declared in the operand.

A byte address is defined as two times the word address of the memory location containing the particular byte. Figure 4-2 illustrates byte addressing for a portion of memory.

## Assembler Instructions

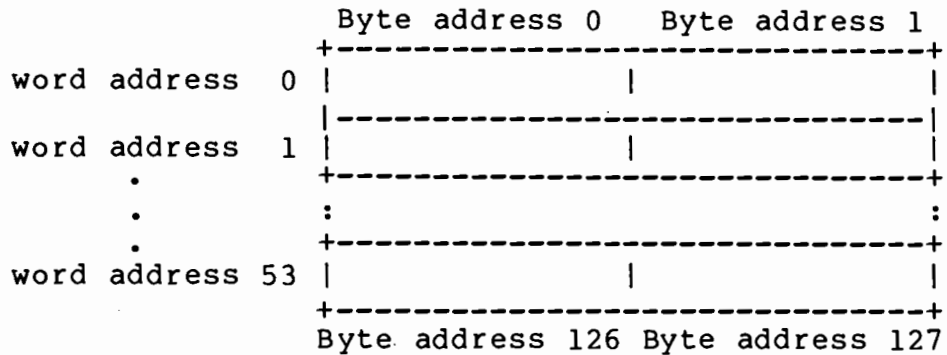


Figure 4-2. Byte Addressing

If the byte location is the left half of the memory location, bit 0 of the byte address is 0; if the byte location is the right half of the memory location, bit 0 is 1.

### NOTE

Take care when using the label of a DBL or DBR pseudo instruction as an indirect address elsewhere in the source program. You must keep track of whether you are using word addresses or byte addresses.

### Example:

```
byte.1 DBL word
byte.2 DBR word
      :
word   BSS 1
```

If 'word' is at the relocatable address 2002B, then 'byte.1' will contain the relocatable value 4004B and 'byte.2' will contain the relocatable value 4005B.

## Assembler Instructions

Example:

Move the bytes in one array to another.

```
address1    DBL byte.array.1    ; Generate a byte address.
byte.array.1  BYT 1,2,3,4,5,6,7,10 ; Define an array.
address2    DBL byte.array.2    ; Define another byte address.
byte.array.2  BSS 4              ; Destination array.
            LDA address1        ; Load the byte address of
            ; the array.
            LDB address2        ; Load the byte address of
            ; the destination array.
            MBT =D8              ; Move 8 bytes. After the
            ; move is complete the A- and
            ; B-Registers contain the byte
            ; addresses of the arrays.
```

Example:

To convert a byte address to a word address:

```
            LDA byte.address    ; Load the byte address.
            INA
            CLE,ERA              ; Shift right.
            ; The A-Register now contains
            ; the word address of the byte.
```



## Declaring Assembly-Time Variables

An assembly-time variable (ATV) must be declared before it is used. For this purpose use one of the ATV manipulation pseudo ops IGLOBAL, ILOCAL, CGLOBAL or CLOCAL. By specifying the number and size of elements, you can declare assembly-time arrays. The value of an ATV can be changed by using a CSET or ISET statement. This section discusses these six statements.

In contrast, a macro parameter can be assigned a value when the macro that references that parameter is called. Once a parameter is defined, it cannot be changed. For more details on macro parameters, refer to Chapter 5.

Assembly-time variables and formal macro parameters begin with an ampersand (&). They can be up to 16 characters in length. The range of integer ATVs is -32768 to 32767. They are denoted by the beginning "&" character.

System assembly-time variables are denoted by the beginning beginning "&." characters. Some system ATVs are available for you to use. These are discussed in Appendix J.

## Substituting Values for Assembly-Time Variables

An assembly-time variable or macro parameter has a value substituted for it everywhere it appears except when it appears in any of the following:

- Column one of an ATV manipulation pseudo op.
- In the comments field, except that of the NAM statement.
- In the macro parameter field of the macro name statement.
- In macro definitions (values will be substituted for the ATV when the macro is called).
- In REPEAT and AWHILE loops (values will be substituted when the loops are expanded).
- Between pairs of single quotes.

## ILOCAL, IGLOBAL, CLOCAL, CGLOBAL

These pseudo ops assign initial value to assembly-time variables. The label is an assembly-time variable. The num and size parameters are optional.

label [num,size]	ILOCAL	integer expression	[:comment]
label [num,size]	IGLOBAL	assembly-time variable	[:comment]
label [num,size]	CLOCAL	character expression	[:comment]
label [num,size]	CGLOBAL	assembly-time variable	[:comment]

num is a single integer value that corresponds to the number of elements in an array. A zero value for num can be present only in CGLOBAL or CLOCAL statements.

size is the size of each element. For type integer, the size is 1 and need not be specified. For character strings, the size is the maximum number of characters in each element. If omitted, the value can never exceed the number of characters of the original value appearing in the operand field. For instance, a variable could be declared as a string that will be set to a longer string later. Default size is 1.

NOTE: If these parameters are input, they must be input using brackets, as shown in the following examples.

Examples of labels:

&status	an assembly-time variable of type character or type integer.
&string[0,10]	a string of up to 10 characters.
&array[3,1]	an array of three elements, each one character long or an array of 3 single integers.
&strings[3,5]	an array of 3 strings, each up to 5 characters long.

The opcode declares the type of the assembly-time variable:

ILOCAL	- type integer and local scope.
IGLOBAL	- type integer and global scope.
CLOCAL	- type character and local scope.
CGLOBAL	- type character and global scope.

## Assembler Instructions

The operand, or the value assigned to an assembly-time variable, can be an integer expression or it can be a character expression.

An assembly-time variable can take on one of two possible scopes: global or local.

A global assembly-time variable can be referenced throughout a module and throughout a multi-module file including the macros called by the module. Its value can be changed, tested or used anywhere in the module, except in macros that declare local assembly-time variables of the same name as the global or have parameters of that name.

Local assembly-time variables can be declared only in macro definitions, REPEAT and AWHILE loops. They are valid only within the macro in which they are defined and within the inner macros called by that macro. If an inner macro declares an assembly-time variable of the same name as one declared in an outer macro, the declaration in the inner macro is effective for that inner macro only.

Example:

```
MACRO
OUTER
&var    ILOCAL 0      ; Declare a local assembly-time variable.
        :             ; Set its initial value to 0.
        LDA =D&var    ; Use &var. Zero gets substituted for it.
        INNER1       ; Call macro INNER1, defined below.
        LDA =D&var    ; &var has same value as above. It has
        :             ; not been changed by macro INNER1.
        INNER2       ; Call macro INNER2, defined below.
        LDA =D&var    ; &var gets a 4 substituted for it.
        :             ; It was changed by INNER2 which declared
        :             ; no &var of its own.
ENDMAC

;
MACRO
INNER1
&var    ILOCAL 1      ; Declare &var of value 1.
        LDA =D&var    ; Use &var - a one gets substituted for it.
ENDMAC

;
MACRO
INNER2
&var    ISET    4      ;Change the value of &var declared in OUTER.
ENDMAC
```

## Assembler Instructions

In this example, two different assembly-time variables by the name &var are used. In macro OUTER, the value that &var had before INNER1 is the same value it has after INNER1 is called. The value declared for &var in INNER1 is only effective for the macro INNER1. However, in the macro INNER2, no assembly-time variable by the name of &var was declared. Therefore, the value of the &var declared in OUTER will change after INNER2 is executed.

Assembly-time arrays are initialized by listing integer or character expressions separated by commas. A count parameter surrounded by square brackets is optionally used to indicate a repetition of initial values.

Example:

```
&Y          IGLOBAL  10
&X[10,1]    IGLOBAL  [3]7,[2]3232,12,101B,[2]-&Y+8,10
```

In this example, &X is declared to be a global array of type integer. The values are: 7,7,7,3232,3232,12,101B,-2,-2,10.

To reference an element of an assembly-time array, designate only that element. For instance, to reference the sixth element of the array defined above, specify &X[6].

Example:

```
&X[6] ISET 7
```

A type-character assembly-time variable is assigned a value by setting it to a character string. A character expression is a set of character strings or assembly-time variables concatenated together.

Examples:

```
&Z[5,3]    CGLOBAL 'abc','def','ghi','jkl','mno'
           ; &Z contains five 3-character strings
&A          CLOCAL 'aaa'
&B          CLOCAL &A'bbb' ; &B contains 'aaabbb'
```

**ISET, CSET**

```
label ISET integer expression [;comment]
```

```
label CSET character expression [;comment]
```

The ISET and CSET pseudo ops change the value of a type-integer or type-character assembly-time variable.

The label is a legal assembly-time variable symbol that was previously declared with an ILOCAL, IGLOBAL, CLOCAL, or CGLOBAL pseudo op.

The operand is an integer expression (for ISET) or a character expression (for CSET).

**Examples:**

```
&P1 IGLOBAL 1 ; &P1 is declared to be value 1.
&P1 ISET 8+&P1 ; Change the value of &P1 to 9.

&c1[0,5] CLOCAL '0' ; Reserve space for 5 characters.
&c1 CSET 'abcde' ; Change the value of &c1 to 'abcde'.
```

The assembler performs string substitution for the entire operand field, substituting assembly-time variables and macro parameters for the actual values. See the beginning of this section for rules on when to substitute what value. The operand is then interpreted as an integer expression or a character string.

**Example:**

```
&A IGLOBAL -12 ; Declare an integer ATV.
&B ILOCAL 0 ; Declare another integer ATV.
&B ISET 4+(:L:&A) ; &B is changed to 4+3 (length of &A).
```

## Expressions Using Assembly-Time Variables

An expression is a combination of terms and operators that can be resolved to a value. There are several types of operators that can be used to form expressions in Macro/1000:

```

unary operators      -, :NOT:, :L:, :T:, :S:, and :UC:
arithmetic operators *, /, +, -, :ROT:, :MOD:, :LSH:, :ASH:
comparison operators =, >=, <=, <>, >, and <
logical operators   :AND:, :OR:
    
```

Concatenation is also an operation in Macro/1000.

The operators :L:, :T:, :S: and :UC:, are called attributes. Their operations are performed at assembly-time and are used in the operand field of the ISET and CSET pseudo ops.

### Unary Operators

The unary operators are:

```

:NOT: - (negate) :L: (length) :S: (substring) :T: (type)
:UC: (upper case)
    
```

:NOT:

The :NOT: operator performs a logical negation on a number; that is, each bit in the representation of the number is complemented (ones complement).

Example:

```

&NUM IGLOBAL 0 ; Declare ATV and initialize to 0.
&NOTNUM IGLOBAL :NOT:&NUM ; &NOTNUM is declared and
AIF (:NOT:&P1)<3 ; initialized to -1.
    
```

## Assembler Instructions

### Negate (-)

The negate operand (-) causes an arithmetic negation (two's complement) of a number.

### Examples:

```
        CPA =B-10
length DEF =D-8
```

### Length Attribute (:L:)

The length attribute is replaced by the length of the string contained in the assembly-time variable that follows it. An integer always results from the use of the length attribute.

If you use a type integer assembly-time variable with this attribute, then the result is the number of significant digits in its value.

### Example:

```
&LENGTH IGLOBAL 0
&LENGTH ISET :L:&MESSAGE ; &LENGTH is set to number of
                          ; characters in &MESSAGE.

      JSB EXEC
      DEF  *+5           ; return address
      DEF  =D2           ; EXEC 2 - output
      DEF  =D1
      DEF  =S&MESSAGE   ; string to be output
      DEF  =L-&LENGTH   ; negative number of characters in
                          ; string.
```

## Assembler Instructions

### Type Attribute (:T:)

The type attribute is replaced by the current type of the assembly-time variable that follows it. A character string always results from use of the type attribute. The possible types are:

- I - type integer
- C - type character
- U - undeclared

### Examples:

```
&TYPE  CGLOBAL ' ' ; Declaration of &TYPE
&INT   IGLOBAL '0' ; Declaration of integer &INT
&TYPE  CSET :T:&INT ; The character ATV &TYPE is set to
                    ; the character 'I' since &INT is
                    ; of type integer.
```

```
        AIF :T:&XYZ='u'
&XYZ   CGLOBAL 'XXX'
        AELSE
&XYZ   CSET 'XXX'
        AENDIF
```



## Assembler Instructions

### Substring Attribute (:S:)

The substring attribute is replaced at assembly time by a portion of a type-character assembly-time variable, macro parameter, or string. The attribute looks like this:

```
:S:[start,num]&atv
```

where:

start is the relative place of the starting character.

num is the number of characters desired.

&atv is a type character assembly-time variable.

Start and num can be integers or assembly-time variables of type integer. They cannot be ATV array references or expressions. Start must be positive and less than or equal to the length of &ATV.

For instance, :S:[2,7]&string means starting with the second character of &string, pull out the next seven characters to make a substring.

Example:

```
&substring[0,19] CGLOBAL 'XXXXXXXXXX' ; declare variable
; to hold substring.
&string          CGLOBAL 'Macro Assembler' ; declare string
&substring       CSET   :S:[7,9]&string ; substring is now
; "Assembler".
```

### Upper-Case Attribute (:UC:)

The upper-case attribute maps a character string to all upper case. It precedes a character string or assembly-time variable. The assembler substitutes upper-case letters for any lower-case letters encountered in the string. Special characters do not change.

Example:

```
&upper.case[0,5] CLOCAL '0' ; declaration of ATV to hold upper
; case string.
&lower.case      CLOCAL 'Think'
&upper.case      CSET :UC:&lower.case ; ATV now contains 'THINK'
```

### Arithmetic Operators

There are eight arithmetic operators:

*	:ROT:
/	:MOD:
+	:LSH:
-	:ASH:



When appearing in the operand field as operators, the arithmetic operators \*, /, + and - perform multiplication, division, addition and subtraction on numerical values.

:ROT:

The :ROT: operator rotates the word used to represent a value. It is used in expressions of the form:

```
value :ROT: number
```

where:

value is the value to be rotated.

number is the number of bits to be rotated. A negative value specifies rotate right; a positive value specifies rotate left.

Example:

```
&ROT IGLOBAL 7 ; &ROT is declared and rotated right  
&ROT ISET &ROT:ROT:-2 ; two bits.
```

## Assembler Instructions

:MOD:

The :MOD: operator calculates the modulus of a value. It is used in expressions of the form:

```
number :MOD: divisor
```

where:

number and divisor are integer values.

The value of the expression is the remainder when number is divided by divisor.

Example:

```
&RESIDUE  IGLOBAL 37 ; Declare and initialize &RESIDUE.
&MODULUS  IGLOBAL 5  ; Declare and initialize &MODULUS.
&REL.POS  IGLOBAL &RESIDUE:MOD:&MODULUS ; Declare &REP.POS
                                                ; and initialize
                                                ; to 37 MOD 35
                                                ; (value of 2).
```

:LSH: and :ASH:

The :LSH: operator performs a logical shift and the :ASH: operator performs an arithmetic shift. They are used in expressions of the form:

```
value :ASH: number
value :LSH: number
```

where:

value is the value of the word to be shifted.

number is the number of bits to be shifted in the word, positive for a left shift, negative for a right shift.

Examples:

```
&NUM1  IGLOBAL 100005B      ; Declaration of &NUM1
&NUM1  ISET  &NUM1:ASH:3    ; &NUM1 now equals 100050B.
&NUM2  IGLOBAL 100005B      ; Declaration of &NUM2.
&NUM2  ISET  &NUM2:LSH:3    ; &NUM2 now equals 000050B.
```

## Assembler Instructions

### Comparison Operators

The comparison operators are:

```
= (equal to)
>= (greater than or equal to)
<= (less than or equal to)
<> (not equal to)
> (greater than)
< (less than)
```

Operators = and <> serve for both integer and string comparison. The Assembler determines which type of comparison to perform by the type of the first operand encountered in the expression. For example, if the first operand is a string, the remaining operands will be interpreted as strings, also. Similarly, if the first operand is an integer, the Assembler will interpret all other operands as type integer.

The result of a comparison operand is 1 if the comparison is true, or 0 if it is false. In this way the logical operations can be applied to comparison operations.

Examples:

```
AIF    &X=10
      :
      :
AELSEIF &X=100
      :
      :
AENDIF
```

## Assembler Instructions

### Logical Operators

The logical operators are:

```
:AND:    (logical AND)
:OR:     (logical OR)
```

These operators make comparisons and perform 16-bit logical operations. The result of a logical operation is 1 if true, and 0 if false.

Examples:

```
AIF      (&X>=0) :AND: (&X<10)
      :
AELSEIF  (&X>=10) :AND: (&X<20)
      :
AENDIF
```

### Concatenation

The Assembler substitutes the actual value of an assembly-time variable for every occurrence of it in a label, opcode or operand. This is called string substitution. In Macro/1000, a string is defined as a set of characters, sometimes surrounded by single quotes. Type-character assembly-time variables are symbols representing character strings. When two character strings, or their symbols, are placed immediately adjacent to each other, concatenation occurs.

The Assembler removes all single quotes (except two in a row) in a string before it tries to ascertain meaning from the statement.

Concatenation is allowed anywhere in a program, including macro definitions.

Example:

```
&string1 CLOCAL 'This string'
&string2 CLOCAL 'that one'
&string3 CLOCAL &string1' joined with '&string2
```

Now &string3 contains "This string joined with that one".

## Assembler Instructions

Example:

```
&word1    CLOCAL  'one and'  
&word2    CLOCAL  ' two'  
&word1+2  CLOCAL  &word1&word2 ; &word1+2 contains "one and two"
```

Example:

```
&reg      CLOCAL  'A'  
          LD&reg  address      ; instruction assembled as LDA.
```

This example illustrates string substitution capabilities. No quotes are needed for the characters LD because they appear on the left side of the second string.

Characters appearing to the right of an assembly-time variable must be in single quotes:

```
&string   CLOCAL  &word'right side of ATV'
```

Example:

```
&num      ILOCAL  2  
          LDA     =D&num
```

This example illustrates the capability of using string substitution with literals.

## Conditional Assembly

Conditional assembly pseudo ops are commands to the assembler telling it:

- to either ignore or assemble a set of statements depending on some condition (AIF, AELSEIF, AELSE, AENDIF).
- to assemble a set of statements while some condition is true (AWHILE, AENDWHILE).

Also discussed in this section is MNOTE, the pseudo op to declare user-defined errors, and REPEAT AND ENDREP, which allow you to assemble a set of instructions a specific number of times.

The format for AIF, AELSEIF, and AWHILE pseudo ops is:

```
opcode <operand> [;comment]
```

The format for AELSE, AENDIF, and AENDWHILE pseudo ops is:

```
opcode [;comment]
```

### AIF, AELSEIF and AWHILE Operands

The operand of AIF, AELSEIF, AWHILE consists of:

- assembly-time variables
- integer constants
- characters

and operators:

- |              |                            |                          |
|--------------|----------------------------|--------------------------|
| - unary      | - (negate)                 | :NOT: (logical negate)   |
|              | :L: (length attribute)     | :T: (type attribute)     |
|              | :S: (substring)            | :UC: (upper case)        |
| - arithmetic | * (multiply)               | :ROT: (rotate)           |
|              | / (divide)                 | :MOD: (modulo function)  |
|              | + (add)                    | :LSH: (logical shift)    |
|              | - (subtract)               | :ASH: (arithmetic shift) |
| - logical    | :AND: (logical AND)        | :OR: (logical OR)        |
| - comparison | = (equal)                  | <> (not equal)           |
|              | >= (greater than or equal) | > (greater than)         |
|              | <= (less than or equal)    | < (less than)            |

## Assembler Instructions

The operand can be either a character or an integer expression. The type of the first element determines the type of the expression.

The following operators can be used with integer expressions:

```
<= >= < > * / + -
:ROT: :LSH: :ASH: :MOD:
:AND: :OR: :NOT:
```

The following can be used with either integer or character expressions.

```
:T: :L: :S: = <>
```

### Evaluation of Expressions

The operand of the AIF, AELSEIF, and AWHILE pseudo ops must evaluate to an integer 0 or 1. A non-zero value is interpreted as true; zero value is interpreted as false. In this case, the boolean operations can be applied to comparison operations.

Examples:

```
AIF 1 ; Unconditional true.
AELSEIF &X=5 ; If &X is 5, the expression &X=5 is evaluated
; as value 1 and is therefore true.
AWHILE (&X=10):OR:(&Y=5)
```

In the last example, the comparison result of &X=10 (0 or 1) is logically OR'ed with the comparison result of &Y=5 (0 or 1), giving a 0 or 1 final value to the operand.

More legal operands:

```
AWHILE -5>=(&X+1)
AELSEIF (:L:&P1=12):AND:(&P2='string')
```



## Assembler Instructions

Examples of illegal operands:

```
AIF  &X=+5                ; Plus is not a unary operator.  
Awhile (&X+2=(&Y+4)       ; Unmatched parenthesis.
```

### Using AIF and AELSEIF

If the operand field of an AIF or AELSEIF statement evaluates to a non-zero value, MACRO assembles all of the code following it until an AELSE, AELSEIF or AENDIF is encountered.

If the operand field of an AIF or AELSEIF statement evaluates to a value 0, and an AELSE or AELSEIF is present, MACRO ignores all of the code between the two statements. Assembly will begin at the AELSE or AELSEIF statement. If an AELSE or AELSEIF is not present, MACRO ignores all of the code up to the AENDIF statement.

An AIF statement must appear before any AELSE, AELSEIF, or AENDIF statements. Only one AELSE may appear after an AIF or AELSEIF. There must be one and only one AENDIF for each AIF statement.

Nesting of AIF statements is permitted to 16 levels.

## Assembler Instructions

### Example 1:

```
AIF  &debugflag = 'ON' ; If flag is on, then assemble the call
    WRITE templ,temp2  ; to the macro WRITE.
AENDIF
```

### Example 2:

Depending on the value of the &status, one of the following sections of code will be assembled:

```
AIF      &status=1           : &status is one,
    TYPECHECK P1,P2        ; call macro TYPECHECK.
AELSEIF  &status=2           ; &status is two,
    JMP NEXT.SECTION      ; jump to next section.
AELSEIF  &status=3           ; &status is three,
    OUT.BUFFER buffer     ; call macro OUT.BUFFER
AELSE    ERROR &status      ; If &status is not 1, 2, or 3
                                ; assemble the call to macro ERROR.
AENDIF
```

## Using AWHILE

MACRO will continue to assemble the lines of code between AWHILE and AENDWHILE statements until the operand of the AWHILE is evaluated to a 0 (false condition). Make sure that the operand of the AWHILE will eventually evaluate to 0 or the assembler will be in an infinite loop.

AWHILE statements can be nested to a level of 5 deep.

Examples:

```

asciitab EQU *
&X      IGLOBAL 100B
        AWHILE &X < 132B
&X      ISET &X+1
        OCT &X
        AENDWHILE

```

is the same as typing:

```

asciitab OCT 101
        OCT 102
        OCT 103
        :
        OCT 132

```

If a local assembly-time variable is declared inside an AWHILE loop, it is local only to that loop; it is not known to the code outside of the loop.

You may use the system assembly-time variable, `&.REP` in AWHILE loops. `&.REP`'s original value is 1, and is incremented by 1 each time the loop is repeated.

**Using REPEAT and ENDREP**

```
REPEAT  #repetitions  [;comment]
```

```
ENDREP [;comment]
```

The REPEAT pseudo op commands the assembler to repeatedly assemble a set of instructions a fixed number of times. The set of instructions is terminated by the ENDREP statement.

The operand is an integer expression giving the number of times the set is to be repeated. The integer expression may include assembly time variables and macro parameters.

REPEAT loops may be nested to a level of five deep.

Example:

```
REPEAT  &X+1
DEC     -1
DEC     0
ENDREP
```

If &X=2, this is what would be assembled:

```
DEC     -1
DEC     0
DEC     -1
DEC     0
DEC     -1
DEC     0
```

The maximum combined depth of REPEAT and AWHILE loops is five. For example:

```
AWHILE &>true
  REPEAT 2
    REPEAT 3
      REPEAT 4
        AWHILE &R5
          AENDWHILE
        AENDREP
      ENDREP
    ENDREP
  AENDWHILE
```

If a local assembly-time variable is declared inside a REPEAT loop, it is local only to that loop; it is not known to the code outside the loop.

You can use the system assembly-time variable, `&.REP` in REPEAT loops. `&.REP`'s original value is 1, and is incremented by 1 each time the loop is repeated.

## Using MNOTE

```
MNOTE string expression
```

The MNOTE pseudo op allows you to create user-defined errors by flagging the line as an error, causing an error message to be listed with the list file and incrementing the MACRO error count. The string expression is the message to be listed. The constraints on the string expression are the same as those for the CGLOBAL command.

Example:

```
AIF (&REG <>'A') :AND=(&REG<>'B')
  MNOTE 'Register should be A or B, not '&REG
AELSE
:
AENDIF
```

The following might appear in the listing:

```
MNOTE Register should be A or B, not Q
      ^
```

```
60>> user-defined error
```

# Chapter 5

## Using Macros

The Macro/1000 macro language is the set of statements that allow you to define and access macros as well as create macro libraries. A macro is a representation of a sequence of instructions called a macro definition. When a macro call statement is encountered at assembly-time, it is replaced by the instructions in the macro definition. A macro may be called by many programs and many times within a program.

Macros are different from subroutines. A macro is replaced by its expanded form at assembly-time. Therefore the code is generated for the statements of a macro definition every time the macro is called. Code for a subroutine is generated once and causes a break in program flow at execution time.

A macro must be defined before it is called in a program. You can define a macro by including the definition in the program, by referencing an INCLUDE file, by declaring a macro library, or by calling another macro that provides the definition. Macros can be nested to any level within macro definitions.

Topics covered in this chapter are:

- \* Example of a macro.
- \* Calling Macros.
- \* Writing Macro Definitions. Macro statement formats, special considerations of comments and listing options within the definition, redefinition of opcodes.
- \* Macro Parameters. How to pass and receive information to and from macro calls; formal, actual, and default parameters.
- \* Nesting Macros.
- \* Creating Macro Libraries. How to create and access macro libraries.

## Example of a Macro

Suppose you would like to move the contents of one memory location to another location and you do not want to write the instructions every time you want the move. You can define and call a macro to eliminate the repetition of the instructions. The macro definition in this case would be:

```
1      MACRO
2      MOVEM  &from,&to
3          LDA  &from
4          STA  &to
5      ENDMAC
```

The MACRO statement (1) designates the start of a macro definition.

The second statement is the macro name statement. MOVEM is the name you have given to the macro. It uses two parameters, &from and &to. These are called formal macro parameters and will be replaced by actual values when the macro is invoked. Macro parameters are discussed later in this chapter.

The body of the macro definition (3 and 4) contains two statements; the LDA and STA instructions. The operands are the formal parameters. They, too, will be replaced with actual values when the macro is invoked. The body of a macro can contain any number of statements.

The ENDMAC statement (5) terminates a macro definition.

After the macro has been defined, it can be invoked with a macro call statement such as:

```
MOVEM  address1,address2
```

The code that Macro/1000 will generate is this:

```
LDA  address1
STA  address2
```

By convention, macros are defined at the beginning of a source file; however, definitions can actually be anywhere in the source code, provided the macro is defined before it is called.

## Calling Macros

The format of the macro call statement is:

```
[label]    name    [parameter list]    [;comments]
```

The label is an optional parameter and is treated in the same way as a formal parameter, if it is defined in the macro.

The name identifies the macro to be called, as specified in the macro definition.

The parameter list can contain zero or more actual parameters. These parameters will replace the formal parameters of the macro definition when the macro is expanded. Formal and actual parameters are discussed later in this chapter.

Actual parameters are treated as character strings and their usage is determined by the macro definition.

The macro call statement always appears in the listing and the expanded code appears only if in long or medium listing mode.

From the example above, the macro call statement is:

```
MOVEM    address1,address2
```

where MOVEM is the name of the macro definition, and address1 and address2 make up the parameter list.



## Using Macros

### Using Macro Libraries

The macro to be called may have been previously defined in the source code, or its definition may be contained in a macro library.

Before a macro contained in a macro library can be accessed, a MACLIB statement which references that library must appear in the program. This informs the Macro Assembler to search that library if a macro which has not been defined in the program is encountered.

The format of the MACLIB statement is:

```
MACLIB namr
```

No label is required. If specified, it is ignored.

The operand field contains an RTE file NAMR. The file NAMR can be represented by a single assembly-time variable, a macro parameter, or can be entered explicitly. No line-continuation markers are permitted and, since no concatenation is performed on the statement, only one assembly-time variable is permitted.

Once you have entered the MACLIB statement, you can call any macros contained in the same library.

If a macro that has been defined in the source code has the same name as a macro contained in the macro library, then any calls to that macro will refer to the macro defined in the source code. However, you cannot call a library macro and then define another macro that has the same name.

Libraries will be searched in the order they are referenced by MACLIB statements within the source code.

## Writing Macro Definitions

A macro definition is a set of statements consisting of a MACRO statement, a macro name statement, the macro body and the ENDMAC statement.

### The MACRO Statement

The MACRO statement indicates the beginning of the macro definition, and is the first statement of every macro definition. The format is:

```
MACRO [n,n,n]
```

The label field is ignored.

The operand field is optional and specifies in which columns the fields will begin when the macro is expanded. If an operand appears in the statement, it must contain three integers, separated by commas, that indicate the starting columns of the opcode, operand and comment field, respectively.

No error will result if the substituted value crosses a field boundary during a macro expansion. When a field, as specified by the parameters, is not large enough, it will be extended and other fields may be shifted to accommodate the change. For example, if the opcode field is specified to start at column 5 and the label is 6 characters long, the opcode field will begin at column 8.

## The Macro Name Statement

The macro name statement assigns a symbolic name to a macro through which the macro can be referenced. This statement also defines the list of formal macro parameters.

It must be the second statement of every macro definition.

The format of this statement is:

```
[parameter]  name  [parameter list] [;comments]
```

If a label parameter is used, it must begin in column one.

A macro name must start with an alphabetic character or a period (.) and can be followed by one to 15 alphanumeric characters. If a macro is being defined within the body of another macro, the macro name can be an assembly-time variable or a macro parameter.

The parameter list is a set of one or more macro parameters and their optional default values, separated by commas.

The parameters included in a name statement, in both the label parameter and parameter list, are called formal parameters. The formal parameters are assigned values when the macro is called.

Macro parameters are discussed in more detail in the following section of this chapter.

## Using Macros

### Redefinition of Opcodes

An opcode can be redefined as a macro by using the opcode as a macro name in a macro definition. For example:

```
MACRO
  MPY      &PAR1
          JSB      MULTI
          DEF      MULTIRTN&.Q
MULTIRTN&.Q  DEF      &PAR1
          EQU      *
          ENDMAC
```



To use a redefined opcode as the actual opcode (and not as the macro you changed it to) use the `:OP:` operator. This is a unary operator used in the opcode field, preceding the opcode. It tells the assembler that the characters that follow should be interpreted as an opcode, not as a macro.

For example, to use the `MPY` opcode after you have defined it:

```
LABEL :OP:MPY VALUE
```

The `:OP:` operator applies to the entire opcode field. If `:OP:` appears before a line containing several opcodes (from the alter-skip or shift-rotate group), all will be interpreted as regular opcodes, even though several may have been redefined as macros.

If more than one opcode appearing on a line has been redefined as a macro, and no `:OP:` operator is used, the first opcode will be expanded as a macro, and the remaining opcodes will be ignored. For example, if the opcodes `CMA` and `INA` have been redefined as macros, the statement:

```
:OP:CMA,INA
```

would cause both opcodes to be treated as regular opcodes. However, the statement:

```
CMA,INA
```

will cause the macro `CMA` to be expanded, and the opcode `INA` to be ignored.

## The Macro Body

The macro body is one or more assembly-language statements that are generated each time a macro is called.

The following is an example of a macro definition:

```

        MACRO 7,12,21      ;MACRO statement
&label MOVE &from,&to    ;Macro name statement
&label LDA  &from        ;Macro body statement
        STA  &to         ;Macro body statement
        ENDMAC          ;ENDMAC statement

```

The macro name statement in this example indicates that the macro is to be invoked with three parameters in the parameter list (one is a label parameter).

The macro body can use these parameters to perform its actions.

When the macro is called, the formal parameters &label, &to and &from in the body will be replaced by the values specified in the call.

For example, if the macro call statement:

```
HERE  MOVE  ADDR1,ADDR2
```

is encountered, the formal parameters &label, &from and &to are replaced by the symbols HERE, ADDR1 and ADDR2. The assembly language statements generated are:

```
HERE  LDA  ADDR1
      STA  ADDR2
```

If the following macro call statement is encountered:

```
LL32  MOVE  ADDR1+OFFSET,ADDR2+OFFSET
```

the assembly language statements generated are:

```
LL32  LDA  ADDR1+OFFSET
      STA  ADDR2+OFFSET
```

## Using Macros

It is important to remember that labels occurring in statements in the macro body are generated each time the macro is expanded. To avoid having the same label generated each time the macro is expanded, system assembly-time variables can be used to generate unique labels. System assembly-time variables are discussed in Appendix J.

Another method of ensuring that a unique label will be generated in each expansion is to define labels in a macro definition as macro parameters such that the unique label names are assigned on each call.

You do not need to declare a label parameter on the macro name statement if you only need to declare a label for the first executable word of the macro. The above examples have labels for illustration purposes. The following example illustrates how to define a label for the first word of a macro.

Example:

```
MACRO
  MOVE  &FROM,&TO
  LDA   &FROM
  STA   &TO
ENDMAC
```

Now call the macro as follows:

```
LABEL MOVE ADDR1,ADDR2
```

or

```
LABEL EQU *
        MOVE ADDR1,ADDR2
```

and any references to LABEL will reference the word containing LDA ADDR1.

**Comments**

A comment statement that starts with an asterisk in column one will appear in the listing along with the other statements that come from the macro definition. A comment statement that starts with a period in column one, immediately followed by an asterisk, will not appear when the macro is expanded.

For example, when the macro MIN is defined, several comments are included before the macro name statement to provide information about the macro. To avoid unnecessary repetition of these comments each time the macro is expanded, they begin with a period followed by an asterisk:

```

MACRO
.******
.*This macro returns the minimum of two numbers*
.*in the A-Register.                               *
.*Creation date: 6/17/81  Date changed: 7/3/81 *
.*WARNING - There is no check for overflow      *
.******
MIN  &P1,&P2
*Compute minimum number
  LDA    &P2
  CMA,INA
  ADA    &P1
  SSA,RSS
  CLA
  ADA    &P2
*Minimum number is now contained in A-Register
ENDMAC

```

When this macro is expanded in a medium or long listing the two comments that are within the body will appear along with the rest of the code being generated.

**The ENDMAC Statement**

The ENDMAC statement signifies the end of a macro definition. It must be the last statement in every macro definition. The format of this statement is:

```
ENDMAC [;comments]
```

## Macro Parameters

Information is passed from a macro call to the macro definition through macro parameters. A formal macro parameter is a symbol in a macro definition that can be assigned values by corresponding actual parameters in a macro call. An actual macro parameter is a value that is passed to a macro definition.

### Formal Macro Parameters

A formal macro parameter appears in the macro name statement and then can be accessed throughout the macro definition. It must start with an ampersand (&) and can be followed by one to fifteen alphanumeric characters.

A macro parameter cannot be changed by an ISET or CSET instruction. It is always of type character.

Formal parameters can appear in the label field of the macro name statement. They are treated as regular formal parameters. Any formal parameter can appear in the label field in the body of the macro.

Valid formal parameters:

&VARIABLE	&l6B	&32767
&Variable	&X25f1	&label

Note that lower case characters are mapped to upper case characters. From the example above, &VARIABLE and &Variable are the same parameters.

Invalid formal parameters:

ADDR1	first character not an ampersand
&XYZ"1"	quotes are illegal
&abcdefghijklmnop	more than 15 characters after ampersand
&Price\$3.40	dollar sign is illegal



## Using Macros

When MACRO encounters an assembly-time variable or macro parameter in a macro expansion, the order of selection to determine what value is to be substituted for the ATV or macro parameter is:

1. Formal macro parameters inside of the macro expansion.
2. Assembly-time variables declared to be local for the pending macro expansion.
3. Assembly-time variables declared to be local in a macro that called this pending macro.
4. Assembly-time variables declared to be global.

Example:

```
MACRO
  TYPE &message
    EXT EXEC
    JSB EXEC
    DEF *+5
    DEF =D2           ;EXEC 2, output
    DEF =D1           ;to LU 1
    DEF =S&message   ;
    DEF =L-:L:&message ;negative # of characters
ENDMAC
```

The macro TYPE prints an ASCII string to the operator's terminal. It has one formal parameter, &message. The parameter tells EXEC the actual string, then tells how many characters are in the string.

Example:

```
MACRO
&label INCRE &address, &increment.value
&label LDA &address
        ADA =D&increment.value
        STA &address
ENDMAC
```

The macro INCRE increments the contents of an address by a specific value. The formal parameters are &label, &address and &increment.value.

## Using Macros

If you were to have a label in the macro body and you were to call macro more than once, you would get a duplicate label name. To avoid duplicate label names, use the system assembly-time variable `&.Q`, which generates a unique number every time the macro is called. Append `&.Q` to a symbol name thus:

```
LABEL&.Q
```

Given that `LABEL&.Q` is a label within a macro, the first time that macro is called, `LABEL0` is generated. The second time, `LABEL1` is generated, and so on. Any references to that label must be within the macro.

### Actual Macro Parameters

An actual macro parameter appears in a macro call statement. It contains 0 to 80 ASCII characters optionally surrounded by quotes. If a parameter contains a

```
comma,  
blank,  
semicolon,  
backslash,  
single quote, or  
ampersand,
```

it must be surrounded by single quotes. A zero-length parameter is represented by two adjacent single quotes. Actual macro parameters are always type character.

Actual parameters can appear in the label field of the macro call statement. They follow all the rules of other actual parameters. The parameters in the label field are treated one of two ways depending on the situation at the macro name statement. If a formal parameter appears in the label field of the macro name, then the parameter is taken purely as a parameter to be used as a label or in an operand in the macro definition. If the definition does not have a formal parameter in the label field of the name statement and a label appears in the label field of the call statement, then it is taken as a label. The label refers to the address of the first instruction of the macro expanded code.

## Using Macros

Valid actual parameters:

increment.value periods are legal

'NEW Num' blanks are legal if the parameter is surrounded by single quotes.

348 numbers are legal; this is interpreted as '348'.

'' character string of length zero.

#one All special characters are legal; parameters containing any special characters listed above must be surrounded by single quotes.

STOP!

ABC

&abc assembly-time variables or macro formal parameters must be previously defined.

'&abc' passing the string '&abc' rather than the variable &abc.

Num,NEW commas are not allowed unless the parameter is surrounded by single quotes; this is interpreted as two parameters.

Example:

```
TYPE 'Hi there'
```

which calls the macro TYPE (defined in the previous section), has an actual parameter, 'Hi there'.

```
&msg CGLOBAL 'Hi there'  
TYPE &msg
```

produces the same results as the example above.

```
&msg CGLOBAL 'Hi there'  
TYPE '&msg'
```

passes the string '&msg' to TYPE, not the contents of variable &msg.

## Using Macros

Example:

```
HERE INCRE NOWORDS,2
```

calls the macro INCRE (defined in the previous section) and asks it to increment the contents of NOWORDS by 2. The actual parameter HERE is used as a label within the macro.

### Default Parameters

Any formal parameter may have a default value appended to it. A default value is used when no actual parameter appears in the macro call statement. For instance, `&value'=D5'` is a formal macro parameter with a default value of `'=D5'`. If there is no default value for a formal parameter on the macro name statement and that parameter is defaulted in the call statement, a zero-length string will be used as the actual parameter.

Example:

```
MACRO
  ADDEM      &addr1,&value'=D5',&addr2
  LDA        &addr1
  ADA        &value
  STA        &addr2
ENDMAC
```

If no value for `&value` appears in the macro call statement the default value `'=D5'` is used.

The macro call statement using the default value must leave a space for it. For example, this is a sample macro call statement for the macro ADDEM:

```
ADDEM field1,,field2
```

The commas are required to indicate the second parameter is defaulted.

## Using Macros

Example:

```
MACRO
&label  MOVE    &from,&to,&REG'A'
&label  LD&REG  &from
        ST&REG  &to
ENDMAC
```

All of these macro call statements will produce the same assembled code:

```
HERE MOVE field1,field2    or
HERE MOVE field1,field2,   or
HERE MOVE field1,field2,A
```

The assembler code generated for all of the above statements is:

```
HERE LDA  field1
        STA field2
```

## Nested Macros

Macros that have been defined within the body of another macro are called nested macros. Macros can be called from within other macros (nested macro calls); therefore, it is possible to write a macro definition that is entirely made up of macro calls.

## Recursion

A macro can invoke itself by being called from within its own definition.

Example:

```

MACRO,l=1
    MACRO
    FACT &num
    AIF &num<>0
&Product ISET &Product*&Num
&New.Num ILOCAL &Num-1
    FACT &New.Num
    AENDIF
    ENDMAC
    Nam USE.FACT
*
* This module exercises macro recursion
*
&Product IGLOBAL 1
    FACT 5
    Dec &Product
    End

```

The macro FACT calculates the factorial of its parameter, by invoking itself using consecutively smaller parameters. The assembly-time variable &Product accumulates the factorial value, and the assembly-time variable &New.Num holds the decremented value used in the recursive call. Figure 5-1 contains a partial listing of a program in which the macro FACT is called with an actual parameter whose value is 5.

When using recursion, it is important to remember that a limit must be reached at which point the recursion must stop. In this example, conditional assembly stops the recursion when the macro is called with a parameter equal to zero.

Note that since macro parameters have only one value associated with them, calling the macro with a different actual value will change the value of the parameter in all levels of recursion.

Cross-recursion can occur if a macro invokes a second macro which, in turn, invokes the first. The same restrictions apply to this type of recursion, and again, you must ensure that the process will stop at some point.

## Using Macros

```

PAGE# 1      Macro/1000 Version      8:51 AM TUE., 1 SEPT, 1981
00001      MACRO,1=1
00002      MACRO
00003      FACT &num
00004      AIF &num<>0
00005      &Product ISET &Product*&Num
00006      &New.Num ILOCAL &Num-1
00007      FACT &New.Num
00008      AENDIF
00009      ENDMAC
00010      Nam USE.FACT
00011*
00012*      This module exercises macro recursion
00013*
00014      &Product IGLOBAL 1
00015 00000      FACT '5'
00015      + AIF '5'<>'0'
00015      +&Product ISET 1*5
00015      +&New.Num ILOCAL 5-1
00015 00000      + FACT '4'
00015      + AIF '4'<>'0'
00015      +&Product ISET 5*4
00015      +&New.Num ILOCAL 4-1
00015 00000      + FACT '3'
00015      + AIF '3'<>'0'
00015      +&Product ISET 20*3
00015      +&New.Num ILOCAL 3-1
00015 00000      + FACT '2'
00015      + AIF '2'<>'0'
00015      +&Product ISET 60*2
00015      +&New.Num ILOCAL 2-1
00015 00000      + FACT '1'
00015      + AIF '1'<>'0'
00015      +&Product ISET 120*1
00015      +&New.Num ILOCAL 1-1
00015 00000      + FACT '0'
00015      + AIF '0'<>'0'
00015      -&Product ISET &Product*&Num
00015      -&New.Num ILOCAL &Num-1
00015      - FACT &New.Num
00015      + AENDIF
00015      + AENDIF
00015      + AENDIF
00015      + AENDIF
00015      + AENDIF
00015      + AENDIF
00016 00000 000170      Dec 120
00017      End
Macro: No errors total

```

Figure 5-1. Recursion Example.

## Creating Macro Libraries

A macro library is a file consisting of macro definitions specially formatted for fast and easy access by Macro. You can create a macro library by putting macro definitions in a file, specifying 'M' in the control statement, and running the file through the Macro Assembler. By referencing the library in a source file with a MACLIB statement, you can access all the macros in that library.

The 'M' option on the control statement tells the assembler to create a macro library. No relocatable code is generated by the assembler in this mode. In this case the third parameter on the run string specifies the name of a macro library and not the relocatable file name.

The 'T' option on the control statement takes on a new meaning when used with the 'M' option ('T' normally means to list the symbol table). It causes a list of all the macro names in the library to be placed in the list file.

The following is a procedure to follow to create a macro library. Some sample macros are provided.



## Using Macros

First of all, create the source file. The following is an example:

```
MACRO,M,L
  MACRO
    STOP          ; macro to call exit.
    EXT EXEC
    JSB EXEC
    DEF *+2
    DEF =D6
  ENDMAC
  MACRO
    INCRA &addr1,&value,&addr2
    LDA &addr1
    ADA &value
    STA &addr2
  ENDMAC
  MACRO
    MOVE &from,&to
    LDA &from
    STA &to
  ENDMAC
```

The source file was created with the name &LIB. Schedule Macro/1000 this way:

```
RU,MACRO,&LIB,1,$LIB
```

Macro will place the macros in the file \$LIB. Any of the macros in that file can be accessed by another program via the MACLIB \$LIB statement.

# Appendix A

## Assembler Error Messages

Assembler error messages are descriptions of errors that Macro/1000 could detect as it assembles a source file. When Macro detects an error, it prints the error number and description on the list device or file. At the end of compilation, Macro prints an error summary including the line numbers where errors occur and the total number of errors encountered.

Error conditions are returned through the P-type global lP. See the Terminal User' Manual for more information on P-type globals.

If lP is greater than zero, macro generates a non-existent external in the current program so the loader will give an undefined external error at load time.

The following is a description of the Macro/1000 error messages in numerical order.

ERROR #	DESCRIPTION
---------	-------------

1	>> Illegal file namr.
---	-----------------------

2	>> INCLUDE files must not be nested more than five deep.
---	--

4	>> Opcode illegal in absolute assembly.
---	---

5	>> Greater than 1/4 million symbols used: cannot give symbol-table dump.
---	--

51	>> Expression in AIF or AELSEIF statement does not result in a 0 or 1.
----	--

52	>> End of file found before AENDIF in AIF statement.
----	--

53	>> AELSE found before AIF: this line gets ignored.
----	--

54	>> AENDIF found outside of AIF statement: this line gets ignored.
----	---

## Assembler Error Messages

ERROR #	DESCRIPTION
55 >>	AELSEIF found after AELSE: this line gets ignored.
56 >>	Only one AELSE allowed per AIF statement: this line gets ignored.
57 >>	Illegal use of AELSEIF: this line gets ignored.
58 >>	AIFs nested past 16 deep: this line gets ignored.
59 >>	IFNs or IFZs cannot be nested: this line gets ignored.
60 >>	User-defined error.
61 >>	XIF found outside of IFN/IFZ statement: line ignored.
62 >>	No corresponding MACRO, REPEAT or AWHILE.
63 >>	Illegal to use ENT and RPL to two-word RPL values.
64 >>	End of file found before AENDWHILE or ENDREP.
101 >>	Assembly-time variable or macro parameter has more than 16 characters.
102 >>	Illegal assembly-time variable name.
103 >>	Syntax error in assembly-time array: &name[dimension,size].
104 >>	ATV array subscript must be an integer greater than zero.
105 >>	Length of string greater than size specified in ATV array: truncated.
106 >>	The "count" field in assembly-time array must be integer > 0.
107 >>	Missing ']' in operand field of assembly-time array.
108 >>	Syntax error in operand field of assembly-time array declaration.
109 >>	Not enough initial values for assembly-time array.
110 >>	Doubly declared assembly-time variable name.

## Assembler Error Messages

ERROR #	DESCRIPTION
111 >>	Label in ISET, IGLOBAL or ILOCAL statement does not start with '&'.
112 >>	Unrecognized '&' variable.
113 >>	ATV used in a ISET or CSET statement has not been defined.
114 >>	ATV is defined as an array but not used as an array.
115 >>	Referencing an element outside the dimension defined by ATV array.
116 >>	String longer than maximum specified in declaration: truncated.
117 >>	Result of ILOCAL or IGLOBAL is not an integer: defaulted to 0.
118 >>	ATV array size must be $\leq 80$ and $> 0$ .
119 >>	Array subscript must be surrounded by square brackets.
120 >>	Array subscript must not itself be an array.
121 >>	Comparison is not allowed in ATV manipulation.
122 >>	Type conflict in ISET or CSET statement: value of ATV is unchanged.
123 >>	Dimension or size of element in ATV array must not be $\leq 0$ .
124 >>	ILOCAL or CLOCAL must be declared inside a macro call.
125 >>	Array subscript must be single integer or integer variable.
126 >>	Size specified in IGLOBAL and ILOCAL ignored: defaulted to one word.
127 >>	Too many elements in ATV array declaration: excess ignored.
128 >>	No operand in CGLOBAL/CLOCAL: defaulted to null string.

## Assembler Error Messages

ERROR #	DESCRIPTION
151 >>	Illegal column indicator on MACRO statement.
152 >>	Macro name missing from macro definition.
153 >>	Macro name can only contain A-Z, a-z, 0-9, or '.'.
154 >>	Macro by this name already defined.
155 >>	ENDMAC statement missing.
156 >>	String must be <= 80 character: string truncated.
157 >>	Illegal formal macro parameter.
158 >>	Default value too long for listing.
159 >>	Formal parameter must start with '&'.
160 >>	Illegal actual macro parameter.
161 >>	Too many parameters for this macro call.
162 >>	Repeats cannot be nested more than five deep.
163 >>	Expression on REPEAT or REP must have positive integer result.
164 >>	Illegal expression on AWHILE statement.
165 >>	Expression on AWHILE must have less than 80 characters.
166 >>	More than 100 EXTRACT/DELETE macros for this file.
167 >>	Do not use both EXTRACT and DELETE following this INCLUDE or MACLIB.
168 >>	Only five macro libraries allowed per program.
201 >>	Mnemonic field missing.
202 >>	Line too long after string substitution.
203 >>	Column indicators must be three integers separated by commas.
204 >>	Mnemonic field longer than 16 characters.

## Assembler Error Messages

ERROR #	DESCRIPTION
205 >>	END statement missing.
206 >>	Mnemonic column must start past column 1.
207 >>	Column indicators must leave room for next field.
208 >>	Comment field must start before column 128.
209 >>	Label longer than 16 characters.
210 >>	Illegal character in label.
211 >>	Illegal character in opcode field.
212 >>	Opcode illegal in this type of assembly.
213 >>	Operand field missing.
214 >>	Opcode not recognized.
215 >>	Undefined symbol.
216 >>	Too many nested parentheses: limit is 10.
217 >>	Incomplete expression in operand field.
218 >>	String encountered in an integer expression, default to 0.
219 >>	RPL label must not be used in operand field.
220 >>	'(' or integer must be preceded by an operator.
221 >>	Syntax error in expression.
222 >>	Integer divide results in overflow.
223 >>	& variable must follow :L:, :S: or :T: operators.
224 >>	Illegal use of :T: operator.
225 >>	:NOT: must be followed by a type integer variable.
226 >>	Syntax error in substring :S:[var,var]string.

## Assembler Error Messages

ERROR #	DESCRIPTION
227 >>	Number in substring must be $\geq 1$ .
228 >>	Length of substring exceeds current length of string.
229 >>	' ) ' encountered without corresponding ' ( '.
230 >>	' ) ' must be preceded by an integer result.
231 >>	Integer exceeds range -32768 to 32767.
232 >>	Substring construct cannot be nested.
233 >>	Substring starting character exceeds string length.
234 >>	Result of expression must be within 0 to 32767.
235 >>	ASCII string in GEN and LOD record must be $\leq 125$ words.
236 >>	Legal string compare operators are = and <>.
237 >>	Line continuation must not start before the operand field.
238 >>	Duplicate label definition.
239 >>	Illegal operator in expression.
240 >>	Operand must be integer or absolute expression.
241 >>	Undefined entry point.
242 >>	Only one operand can be relocatable.
243 >>	Illegal character in expression.
244 >>	Result of an EQU expression cannot be indirect.
245 >>	Illegal floating-point number construct.
251 >>	Illegal column indicator in COL statement.
252 >>	Keyword must be ON, OFF, SHORT, MEDIUM, or LONG.
253 >>	Octal integers cannot contain an 8 or 9.
254 >>	Literals not legal on this opcode.

## Assembler Error Messages

ERROR #	DESCRIPTION
255 >>	Keyword must be PROGRAM, COMMON, SAVE, CODE, or BASE.
256 >>	ORR must appear before this opcode.
257 >>	ORR found before corresponding ORG or ORB.
258 >>	Operand must be absolute or relocatable expression.
259 >>	Variable not found.
260 >>	Legal literals are =D, =B, =F, =A, =L, =R, and =S.
261 >>	Integer expected.
262 >>	String expected.
263 >>	Label missing.
264 >>	Doubly defined entry-point name.
265 >>	Illegal value for entry point.
266 >>	Result of expression must be absolute integer value.
267 >>	Expression contains two different externals.
268 >>	Two consecutive REP statements encountered.
269 >>	End of file encountered following REP statement.
270 >>	Comment field must be separated from operand field by blank or ';'.
271 >>	Expresssion cannot exist in more than one relocatable space.
272 >>	Label ignored.
273 >>	Syntax error in MIC statement.
274 >>	Duplicate name for MICro-code instruction
275 >>	Duplicate NAM statement.
276 >>	Keyword must be EMA, SAVE, or COMMON.
277 >>	MSEG size must be >= 2 and <= 31.



## Assembler Error Messages

ERROR #	DESCRIPTION
278 >>	Syntax error in ALLOC.
279 >>	EMA and ALLOC EMA or MSEG must not be used in the same program.
280 >>	Duplicate EMA statement.
281 >>	Label longer than five characters in EMA statement.
282 >>	Number of pages specified or MSEG size out of range in EMA statement.
283 >>	Syntax error in EMA statement.
284 >>	Result in operand field cannot be type RPL, or EMA.
285 >>	Local EMA label can be used only in a DDEF statement.
286 >>	DBL/DBR cannot be indirect.
287 >>	Illegal opcode combination.
288 >>	Illegal data.
289 >>	Byte value overflow; must be between -377B and 377B.
290 >>	Not enough parameters in microcode call.
291 >>	Literals are not allowed in microcode call.
292 >>	Expression in RAM pseudo op must be between 0 and 377B.
293 >>	Result of expression in DDEF cannot be RPL or indirect.
300 >>	EXT/ENT statement error.
301 >>	Illegal symbol in EXT/ENT.
302 >>	Doubly defined entry point.
303 >>	Illegal character in ALIAS field.
304 >>	Illegal character in INFO field.
305 >>	EXT & ENT must not reference the same symbol.

## Assembler Error Messages

ERROR #	DESCRIPTION
306	>> Info or alias field on reference to existing symbol.
307	>> Number of externals exceeds 2047.
308	>> Too many parameter types in INFO field.
309	>> I/O select code must be absolute, >0, <64.
310	>> COM operand-field error.
311	>> COM allocation must be absolute and greater than zero.
312	>> COM statement contains illegal symbol.
313	>> COM statement legal only in program relocation space.
314	>> RPL names limited to five characters.
315	>> EMA value not allowed here.
316	>> Operand must be positive, absolute, and less than or equal to 16.
317	>> Subhead parameter must be less than 81 characters.
318	>> Name used both for label and for external replacement opcode.
319	>> Illegal program name.
320	>> Only the =F literal is legal on this opcode.
321	>> Only the =S, =D, =B, =A, =R, and =L literals are legal on this opcode.
322	>> Comment field on NAM statement must not exceed 73 characters.
323	>> EQUs must not be negative when 'ASMB' is the control statement.
324	>> Machine instructions, BSS, COM, ORG, RELOC, ORB, must not precede NAM.
325	>> NAM or ORG statement missing.
326	>> Values on =L literal must be previously defined.





# Appendix B

## Summary of Instructions

This Appendix summarizes the machine instructions and pseudo ops of Macro/1000 in the following order:

### Machine Instructions

- Memory Reference Instructions
- Word, Byte and Bit Processing
- No-operation
- Register Reference/Shift-Rotate Group
- Register Reference/Alter-Skip Group
- Extended Instruction Group (Index Register Manipulation)
- Input/Output
- Overflow
- Halt
- Extended Arithmetic Unit
- Floating-Point Instructions
- Dynamic Mapping Instructions

### Pseudo Instructions

- Assembler Control
- Loader and Generator Instructions
- Program Linkage
- Listing Control
- Storage Allocation
- Constant Definition
- Address and Symbol Definition
- Assembly-Time Variable Declaration
- Conditional Assembly
- Macro Definition

The notations used in this section are:

m	- memory address	A	- A-Register
[ ]	- optional portion of field	B	- B-Register
@	- indirect address indicator	E	- E-Register
lsb	- least significant bit (bit 0)	X	- X-Register
		Y	- Y-Register

Refer to the appropriate computer reference manual for the base set of instructions available in each of the processors used in the HP 1000 computer systems.

## Machine Instructions

### Memory Reference Instructions

OPCODE	INSTRUCTIONS	OPERAND FORMAT
ADA	Add to A.	[@]m or literal.
ADB	Add to B.	[@]m or literal.
LDA	Load into A.	[@]m or literal.
LDB	Load into B.	[@]m or literal.
STA	Store from A.	[@]m.
STB	Store from B.	[@]m.
AND	Logical "AND" to A.	[@]m or literal.
CPA	Compare to A, skip if unequal.	[@]m or literal.
CPB	Compare to B, skip if unequal.	[@]m or literal.
XOR	Exclusive "OR" to A.	[@]m or literal.
IOR	Inclusive "OR" to A.	[@]m or literal.
ISZ	Increment, then skip if zero.	[@]m.
JMP	Jump.	[@]m.
JSB	Jump to subroutine.	[@]m.

## Summary of Instructions

### Word, Byte and Bit Processing

OPCODE	INSTRUCTIONS	OPERAND FORMAT
CMW	Compare words; A and B contain addresses of word arrays.	[@]m or literal is number of words to compare.
MVW	Move words; A contains start of source, B contains start of destination.	[@]m or literal is number of words to move.
CBT	Compare bytes; A and B contain addresses of byte arrays.	[@]m or literal is number of bytes to compare.
LBT	Load byte defined in B to lower byte of A.	No operand.
MBT	Move bytes; A contains start of source, B contains start of destination.	[@]m or literal is number of bytes to move.
SBT	Store lower byte of A into byte address defined in B.	No operand.
SFB	Scan array defined by B for upper and lower byte of A.	No operand.
CBS	Clear bits as per mask.	Operand 1- [@]m or literal (mask). Operand 2- [@]m.
SBS	Set bits as per mask.	Operand 1- [@]m or literal (mask). Operand 2- [@]m.
TBS	Test bits as per mask.	Operand 1- [@]m or literal (mask). Operand 2- [@]m.

### No-Operation

NOP No-operation, skip to next.

## Register Reference, Shift/Rotate Group

### OPCODE INSTRUCTIONS

ALF	Rotate A left four bits.	(No operands in this group.)
BLF	Rotate B left four bits.	
ELA	Rotate E and A left one bit.	
ELB	Rotate E and B left one bit.	
ERA	Rotate E and A right one bit.	
ERB	Rotate E and B right one bit.	
RAL	Rotate A left one bit.	
RAR	Rotate A right one bit.	
RBL	Rotate B left one bit.	
RBR	Rotate B right one bit.	
ALR	Shift A left one bit, clear sign, clear lsb.	
ALS	Shift A left one bit, clear lsb.	
ARS	Shift A right one bit, extend sign, sign unaltered.	
BLR	Shift B left one bit, clear sign, clear lsb.	
BLS	Shift B left one bit, clear lsb.	
BRS	Shift B right one bit, extend sign, sign unaltered.	
CLE	Clear E.	
LAE	Copy lsb of A to E, A is unchanged.	
LBE	Copy lsb of B to E, B is unchanged.	
SAE	Copy sign of A into E, A is unchanged.	
SBE	Copy sign of B into E, B is unchanged.	
SLA	Skip if lsb of A is zero.	
SLB	Skip if lsb of B is zero.	

## Summary of Instructions

Shift/Rotate instructions can be combined as follows:

A-Register Instructions		B-Register Instructions	
$\left( \begin{array}{l} \text{ALS} \\ \text{ARS} \\ \text{RAL} \\ \text{RAR} \\ [\text{ALR}] \\ \text{ALF} \\ \text{ERA} \\ \text{ELA} \\ \text{SAE} \\ \text{LAE} \end{array} \right)$	$\left\{ \begin{array}{l} [\text{,CLE}] \\ [\text{,SLA}] \end{array} \right\}$	$\left( \begin{array}{l} \text{,ALS} \\ \text{,ARS} \\ \text{,RAL} \\ \text{,RAR} \\ [\text{,ALR}] \\ \text{,ALF} \\ \text{,ERA} \\ \text{,ELA} \\ \text{,SAE} \\ \text{,LAE} \end{array} \right)$	$\left\{ \begin{array}{l} [\text{,CLE}] \\ [\text{,SLB}] \end{array} \right\}$
		$\left( \begin{array}{l} \text{BLS} \\ \text{BRS} \\ \text{RBL} \\ \text{RBR} \\ [\text{BLR}] \\ \text{BLF} \\ \text{ERB} \\ \text{ELB} \\ \text{SBE} \\ \text{LBE} \end{array} \right)$	$\left\{ \begin{array}{l} [\text{,BLS}] \\ [\text{,BRS}] \\ [\text{,RBL}] \\ [\text{,RBR}] \\ [\text{,BLR}] \\ [\text{,BLF}] \\ [\text{,ERB}] \\ [\text{,ELB}] \\ [\text{,SBE}] \\ [\text{,LBE}] \end{array} \right\}$

### Register Reference, Alter/Skip Group

#### OPCODE INSTRUCTIONS

CCA	Clear and complement A.	(No operands in this group.)
CCB	Clear and complement B.	
CCE	Clear and complement E.	
CLA	Clear A.	
CLB	Clear B.	
CLE	Clear E.	
CMA	Complement A.	
CMB	Complement B.	
CME	Complement E.	
INA	Increment A by one.	
INB	Increment B by one.	
RSS	Reverse the sense of the skip; if used as a single instruction, unconditional skip the next instruction.	
SEZ	Skip if E is zero.	



## Summary of Instructions

SLA     Skip if lsb of A is zero.  
SLB     Skip if lsb of B is zero.  
SSA     Skip if sign of A is zero.  
SSB     Skip if sign of B is zero.  
SZA     Skip if A is zero.  
SZB     Skip if B is zero.

Alter/skip instructions can be combined as follows:

$$\left\{ \begin{array}{l} \text{CLA} \\ [\text{CMA}] \\ \text{CCA} \end{array} \right\} [,\text{SEZ}] \left\{ \begin{array}{l} ,\text{CLE} \\ [,\text{CME}] \\ ,\text{CCE} \end{array} \right\} [,\text{SSA}] [,\text{SLA}] [,\text{INA}] [,\text{SZA}] [,\text{RSS}]$$
$$\left\{ \begin{array}{l} \text{CLB} \\ [\text{CMB}] \\ \text{CCB} \end{array} \right\} [,\text{SEZ}] \left\{ \begin{array}{l} ,\text{CLE} \\ [,\text{CME}] \\ ,\text{CCE} \end{array} \right\} [,\text{SSB}] [,\text{SLB}] [,\text{INB}] [,\text{SZB}] [,\text{RSS}]$$

## Summary of Instructions

### Extended Instruction Group (Index Register Manipulation)

#### OPCODE INSTRUCTIONS

#### OPERAND FORMAT

ADX*	Add to X.	[@]m or literal.
ADY*	Add to Y.	[@]m or literal.
LAX*	Load A indexed by X.	[@]m.
LAY*	Load A indexed by Y.	[@]m.
LBX*	Load B indexed by X.	[@]m.
LBY*	Load B indexed by Y.	[@]m.
LDX*	Load into X.	[@]m or literal.
LDY*	Load into Y.	[@]m or literal.
SAX*	Store A indexed by X.	[@]m.
SAY*	Store A indexed by Y.	[@]m.
SBX*	Store B indexed by X.	[@]m.
SBY*	Store B indexed by Y.	[@]m.
STX*	Store X.	[@]m.
STY*	Store Y.	[@]m.

CAX\* Copy A to X.

(No operands in this section.)

CAY\* Copy A to Y.

CBX\* Copy B to X.

CBY\* Copy B to Y.

CXA\* Copy X to A.

CXB\* Copy X to B.

CYA\* Copy Y to A.

CYB\* Copy Y to B.

XAX\* Exchange X and A.

XAY\* Exchange Y and A.

XBX\* Exchange X and B.

XBY\* Exchange Y and B.

DSX\* Decrement X by one.

No operand.

DSY\* Decrement Y by one.

No operand.

ISX\* Increment X by one.

No operand.

ISY\* Increment Y by one.

No operand.

JLY\* Jump and load Y.

[@]m.

JPY\* Jump indexed by Y.

m.

\* Implemented in software for L-Series machine.

## Summary of Instructions

### Input/Output, Overflow and Halt

OPCODE	INSTRUCTIONS	OPERAND FORMAT
LIA	Load A with I/O buffer.	Select code.
LIAC	Load A with I/O buffer and clear flag bit.	Select code.
LIB	Load B with I/O buffer.	Select code.
LIBC	Load B with I/O buffer and clear flag bit.	Select code.
MIA	Merge A with I/O buffer.	Select code.
MIAC	Merge A with I/O buffer and clear flag bit.	select code
MIB	Merge B with I/O buffer.	Select code.
MIBC	Merge B with I/O buffer and clear flag bit.	Select code.
OTA	Output A to I/O buffer.	Select code.
OTAC	Output A to I/O buffer and clear flag bit.	Select code.
OTB	Output B to I/O buffer.	Select code.
OTBC	Output B to I/O buffer and clear flag bit.	Select code.
CLC	Clear control bit.	Select code.
CLCC	Clear control and flag bit	Select code.
CLF	Clear flag bit.	Select code.

## Summary of Instructions

SFC	Skip if control bit is zero.	Select code.
SFS	Skip if flag bit is zero.	Select code.
STC	Set control bit.	Select code.
STCC	Set control bit, clear flag.	Select code.
STF	Set flag bit.	Select code.
CLO	Clear 0 bit.	No operand.
SOC	Skip if 0 is clear.	No operand.
SOCC	Skip if 0 is clear, clear 0.	No operand.
SOS	Skip if 0 is set.	No operand.
SOSC	Skip if 0 is set, clear 0.	No operand.
STO	Set 0.	No operand.
HLT	Halt the computer.	select code of flag bit.
HLTC	Halt the computer, clear flag.	Select code.

## Summary of Instructions

### Extended Arithmetic Unit

OPCODE	INSTRUCTIONS	OPERAND FORMAT
DLD	Load A and B.	[@]m or literal.
DST	Store A and B.	[@]m.
MPY	Multiply with A.	[@]m or literal.
DIV	Divide with A and B.	[@]m or literal.
ASL	Arithmetic shift A and B left.	integer, No. bits to shift.
ASR	Arithmetic shift A and B right.	integer, No. bits to shift.
LSL	Logically shift A and B left.	integer, No. bits to shift.
LSR	Logically shift A and B right.	integer, No. bits to shift.
RRL	Rotate A and B left.	integer, No. bits to rotate.
RRR	Rotate A and B right.	integer, No. bits to rotate.
SWP	Swap A and B.	No operand.

## Summary of Instructions

### Floating-Point Instructions

OPCODE	INSTRUCTIONS	OPERAND FORMAT
FAD*	Floating-pt.add to A and B.	[@]m or floating-pt.literal.
FDV*	Floating-pt.divide to A and B.	[@]m or floating-pt.literal.
FIX*	Convert floating to fixed-pt.	No operand.
FLT*	Convert fixed to floating-pt.	No operand.
FMP*	Float pt multiply to A and B.	[@]m
FSB*	Float pt subtract to A and B.	[@]m

\* Implemented in software for L-Series machines.

## Summary of Instructions

### Dynamic Mapping System

OPCODE	INSTRUCTIONS	OPERAND FORMAT
DJP+	Disable MEM and jump.	[@]m.
DJS+	Disable MEM, jump to subroutine.	[@]m.
SJP+	Enable system map and jump.	[@]m.
SJS+	Enable sys map, jump subroutine.	[@]m.
UJP+	Enable user map and jump.	[@]m.
UJS+	Enable user map jump subroutine.	[@]m.
JRS+	Jump and restore status.	operand 1-[@]m or literal. operand 2-[@]m.
LFA+	Load fence from A.	No operand.
LFB+	Load fence from B.	No operand.
PAA+*	Load/store Port A map per A.	No operand.
PAB+*	Load/store Port A map per B.	No operand.
PBA+*	Load/store Port B map per A.	No operand.
PBB+*	Load/store Port B map per B.	No operand.
SYA+	Load/store system map per A.	No operand.
SYB+	Load/store system map per B.	No operand.
USA+	Load/store user map per A.	No operand.
USB+	Load/store user map per B.	No operand.
SSM+	Store status register in memory.	[@]m.

\* Not available in A-Series machines.

## Summary of Instructions

MBF+	Move bytes from alternate map.	No operand.
MBI+	Move bytes into alternate map.	No operand.
MBW+	Move bytes within alt. map.	No operand.
MWF+	Move words from alternate map.	No operand.
MWI+	Move words into alternate map.	No operand.
MWW+	Move words within alt. map.	No operand.
RSA+	Read status register into A.	No operand.
RSB+	Read status register into B.	No operand.
RVA+*	Read violation register into A.	No operand.
RVB+*	Read violation register into B.	No operand.
XCA+	Cross compare A.	[@]m.
XCB+	Cross compare B.	[@]m.
XLA+	Cross load A.	[@]m.
XLB+	Cross load B.	[@]m.
XSA+	Cross store A.	[@]m.
XSB+	Cross store B.	[@]m.
XMA+	Transfer maps internally per A.	No operand.
XMB+	Transfer maps internally per B.	No operand.
XMM+	Transfer map or memory.	No operand.
XMS+	Transfer maps sequentially.	No operand.

+ Not available in L-Series machines.

\* Not available in A-Series machines.



## Pseudo Operations

### Assembler Control

OPCODE	INSTRUCTIONS	OPERAND FORMAT
NAM	Name relocatable program.	Name plus optional parameters.
ORG	Define absolute origin.	Absolute expression.
ORR	Reset program location counter.	No operand.
END	Terminate program.	Name of program starting location.
RELOC	Specify memory space.	Keyword.
INCLUDE	Include a source file in this assembly.	Name of source file.

### Loader and Generator Control

OPCODE	INSTRUCTIONS	OPERAND FORMAT
LOD**	Define loader record.	Number of chars followed by loader record.
GEN**	Define generator record.	Number of chars followed by gen record.

\*\* Not available in M-, E- or F-Series machines.

## Summary of Instructions

### Program Linkage

OPCODE	INSTRUCTIONS	OPERAND FORMAT
ENT	Define entry point.	name=['alias'][,name...]
EXT	Define external routine.	name=['alias'][,name...]
RPL	Replace instruction.	Value of microcode
ALLOC	Allocate memory space.	Keyword.

### Listing Control

OPCODE	INSTRUCTIONS	OPERAND FORMAT
COL	Specify column numbers.	3 integers each a column number.
HED	List heading at top of page.	Heading.
LIST	Alter current state.	Keyword.
SKP	Skip to top of next page.	No operand.
SPC	Skip lines of listing.	No operand.
SUBHEAD	List a subhead at top of page.	Subheading.
SUP	Suppress extended code list.	No operand.
UNS	Resume extended code list.	No operand.

## Summary of Instructions

### Storage Allocation

OPCODE	INSTRUCTIONS	OPERAND FORMAT
BSS	Reserve storage area.	Integer is number of words to reserve.
MSEG	Reserve MSEG size for EMA.	Integer is size in pages.

### Constant Definition

OPCODE	INSTRUCTIONS	OPERAND FORMAT
ASC	Generate ASCII characters.	Number of words, string.
BYT	Define octal byte constants.	Octal constants.
DEC	Define decimal constants.	Decimal constants.
DEX	Define 3-word constants.	Decimal constants.
DEY	Define 4-word constants.	Decimal constants.
LIT	Control placement of literals.	No operand.
LITF	Control placement of literals.	No operand.
OCT	Define an octal constant.	Octal constants.

### Address and Symbol Definition

OPCODE	INSTRUCTIONS	OPERAND FORMAT
DEF	Generate 15-bit address.	[@]m or literal.
DDEF	Generate 32-bit address.	m.
ABS	Define absolute value.	Absolute value.
EQU	Equate value to label.	m.
DBL	Define left byte.	m or literal.
DBR	Define right byte.	m or literal.

## Summary of Instructions

### Assembly-Time Variable Declaration

OPCODE	INSTRUCTIONS	OPERAND FORMAT
CLOCAL	Declare local character ATV.	Character expression.
CGLOBAL	Declare global character ATV.	Character expression.
CSET	Change character ATV.	Character expression.
ILOCAL	Declare local integer ATV.	Integer expression.
IGLOBAL	Declare global integer ATV.	Integer expression.
ISSET	Change integer ATV.	Integer expression.

### Conditional Assembly



OPCODE	INSTRUCTIONS	OPERAND FORMAT
AELSE	AIF construct.	No operand.
AELSEIF	AIF construct.	Assembly-time expression.
AENDIF	End AIF construct.	No operand.
AENDWHILE	End AWHILE loop.	No operand.
AIF	Start AIF construct.	Assembly-time expression.
AWHILE	Start AWHILE loop.	Assembly-time expression.
ENDREP	End REPEAT loop.	No operand.
REPEAT	Start REPEAT loop.	Assembly-time expression.

## Summary of Instructions

### Macro Definition

OPCODE	INSTRUCTIONS	OPERAND FORMAT
MACRO	Start macro definition.	Integers specifying column numbers.
ENDMAC	End macro definition.	No operand.
MACLIB	Specify macro library	Name of macro library.

### Error Reporting

OPCODE	INSTRUCTIONS	OPERAND FORMAT
MNOTE	Note error condition	Character Expression

# Appendix C

## Instructions Index

### SET SUMMARY

The following alphabetic list details the instructions that are available on the various HP 1000 computers. An asterisk in the L-SERIES column indicates that the instruction is implemented in software in RTE-L/XL systems. An "N/A" indicates that the instruction is not available on that model of computer. It is assumed that the M-, E-, F-, and A-SERIES computers contain the Dynamic Mapping System instructions.

INSTRUCTION	M/E/F/A- SERIES	L- SERIES
ABS	Define absolute value.....	yes yes
ADA	Add to A.....	yes yes
ADB	Add to B.....	yes yes
ADX	Add memory to X.....	yes *
ADY	Add memory to Y.....	yes *
AELSE	Conditional assembly.....	yes yes
AELSEIF	Conditional assembly.....	yes yes
AENDIF	Conditional assembly.....	yes yes
AENDWHILE	Conditional assembly.....	yes yes
AIF	Conditional assembly.....	yes yes
ALF	Rotate A left 4.....	yes yes
ALLOC	Allocate memory space.....	yes yes
ALR	Shift A left 1, clear sign.....	yes yes
ALS	Shift left 1.....	yes yes
AND	"And" to A.....	yes yes
ARS	Shift A right 1, sign carry.....	yes yes
ASC	Generate ASCII characters.....	yes yes
ASL	Arithmetic long shift left.....	yes yes
ASR	Arithmetic long shift right.....	yes yes
AWHILE	Conditional assembly.....	yes yes
BLF	Rotate B left 4 bits.....	yes yes
BLR	Shift B left 1, clear sign.....	yes yes
BLS	Shift B left 1.....	yes yes
BRS	Shift B right 1, carry sign.....	yes yes
BSS	Reserve block of storage starting at symbol.....	yes yes
BYT	Defines octal byte constants.....	yes yes

# Instructions Index

INSTRUCTION	M/E/F/A- SERIES	L- SERIES
CAX	Copy A to X.....	yes *
CAY	Copy A to Y.....	yes *
CBS	Clear bits.....	yes *
CBT	Compare bytes.....	yes *
CBX	Copy B to X.....	yes *
CBY	Copy B to Y.....	yes *
CCA	Clear and complement A (1's).....	yes yes
CCB	Clear and complement B (1's).....	yes yes
CCE	Clear and complement E (set E = 1).....	yes yes
CGLOBAL	Assembly-Time variable declaration.....	yes yes
CLA	Clear A.....	yes yes
CLB	Clear B.....	yes yes
CLC	Clear I/O control bit.....	yes yes
CLCC	Clear I/O control bit, clear flag bit....	yes yes
CLE	Clear E.....	yes yes
CLF	Clear I/O flag.....	yes yes
CLO	Clear overflow bit.....	yes yes
CLOCAL	Assembly-Time Variable declaration.....	yes yes
CMA	Complement A.....	yes yes
CMB	Complement B.....	yes yes
CME	Complement E.....	yes yes
CMW	Compare words.....	yes yes
COL	Specify column numbers for list output...	yes yes
COM	Reserve block of common storage.....	yes yes
CPA	Compare to A, skip if unequal.....	yes yes
CPB	Compare to B, skip if unequal.....	yes yes
CSET	Assembly-Time Variable declaration.....	yes yes
CXA	Copy X to A.....	yes *
CXB	Copy X to B.....	yes *
CYA	Copy Y to A.....	yes *
CYB	Copy Y to B.....	yes *
DBL	Define left byte (bits 8-15) address.....	yes yes
DBR	Define right byte (bits 0-7) address.....	yes yes
DDEF	Define address for EMA.....	yes yes
DEC	Define decimal constant.....	yes yes
DEF	Define address.....	yes yes
DEX	Define extended precision constant (3 words).....	yes yes
DEY	Define double-word constant (4 words)....	yes yes
DIV	Divide.....	yes yes
DJP	Disable MEM and jump.....	yes N/A
DJS	Disable MEM and jump to subroutine.....	yes N/A
DLD	Double load.....	yes yes
DST	Double store.....	yes yes
DSX	Decrement X and skip if zero.....	yes *
DSY	Decrement Y and skip if zero.....	yes *

## Instructions Index

INSTRUCTION	M/E/F/A- SERIES	L- SERIES
ELA	Rotate E and A left 1.....	yes yes
ELB	Rotate E and B left 1.....	yes yes
EMA	Extended Memory Area.....	yes N/A
END	Terminate program.....	yes yes
ENDMAC	End macro definition.....	yes yes
ENDREP	End repeat loop.....	yes yes
ENT	Entry point.....	yes yes
ERA	Rotate E and A right 1.....	yes yes
ERB	Rotate E and B right 1.....	yes yes
EQU	Equate symbol.....	yes yes
EXT	External reference.....	yes yes
FAD	Floating add.....	yes *
FDV	Floating divide.....	yes *
FIX	Convert floating-point to fixed-point....	yes *
FLT	Convert fixed-point to floating-point....	yes *
FMP	Floating multiply.....	yes *
FSB	Floating subtract.....	yes *
GEN	Generator record.....	N/A yes
HED	Print heading at top of each page.....	yes yes
HLT	Halt.....	yes yes
HLTC	Halt, clear flag bit.....	yes yes
IFN	When N appears in Control statement, assemble ensuing instructions.....	yes yes
IFZ	When Z appears in Control statement, assemble ensuing instructions.....	yes yes
IGLOBAL	Assembly-time variable declaration.....	yes yes
ILOCAL	Assembly-time variable declaration.....	yes yes
INA	Increment A by 1.....	yes yes
INB	Increment B by 1.....	yes yes
INCLUDE	Include file.....	yes yes
IOR	Inclusive "or" to A.....	yes yes
ISSET	Assembly-time variable declaration.....	yes yes
ISX	Increment X and skip if zero.....	yes *
ISY	Increment Y and skip if zero.....	yes *
ISZ	Increment, then skip if zero.....	yes yes
JLY	Jump and load Y.....	yes *
JMP	Jump.....	yes yes
JPY	Jump indexed by Y.....	yes *
JRS	Jump and restore status.....	yes N/A
JSB	Jump to subroutine.....	yes yes



## Instructions Index

INSTRUCTION	M/E/F/A- SERIES	L- SERIES	
LAE	Copy low-order bit of A into E.....	yes	yes
LAX	Load A from memory indexed by X.....	yes	*
LAY	Load A from memory indexed by Y.....	yes	*
LBE	Copy low-order bit of B into E.....	yes	yes
LBT	Load byte.....	yes	*
LBX	Load B from memory indexed by X.....	yes	*
LBY	Load B from memory indexed by Y.....	yes	*
LDA	Load into A.....	yes	yes
LDB	Load into B.....	yes	yes
LDX	Load X from memory.....	yes	*
LDY	Load Y from memory.....	yes	*
LFA	Load fence from A.....	yes	N/A
LFB	Load fence from B.....	yes	N/A
LIA	Load into A from I/O channel.....	yes	yes
LIAC	Load into A from I/O channel; clear flag bit.....	yes	yes
LIB	Load into B from I/O channel.....	yes	yes
LIBC	Load into B from I/O channel; clear flag bit.....	yes	yes
LIST	Specify list option.....	yes	yes
LIT	Control placement of literals.....	yes	yes
LITF	Control placement of literals.....	yes	yes
LOD	Loader record.....	N/A	yes
LSL	Logical long shift left.....	yes	yes
LSR	Logical long shift right.....	yes	yes
LST	Resume list output (follows a UNL).....	yes	yes
MACLIB	Specify macro library.....	yes	yes
MACRO	Start macro definition.....	yes	yes
MBF	Move bytes from alternate map.....	yes	*
MBI	Move bytes into alternate map.....	yes	*
MBT	Move byte.....	yes	*
MBW	Move bytes within alternate map.....	yes	*
MIA	Merge (or) into A from I/O channel.....	yes	yes
MIAC	Merge (or) into A from I/O channel, clear flag bit.....	yes	yes
MIB	Merge (or) into B from I/O channel.....	yes	yes
MIBC	Merge (or) into B from I/O channel, clear flag bit.....	yes	yes
MIC	Define user instruction.....	yes	yes
MNOTE	Note error condition.....	yes	yes
MPY	Multiply.....	yes	yes
MSEG	Declare mseg size.....	yes	yes
MVW	Move words.....	yes	*
MWF	Move words from alternate map.....	yes	*
MWI	Move words into alternate map.....	yes	*
MWW	Move words within alternate map.....	yes	*

## Instructions Index

INSTRUCTION		M/E/F/A- SERIES	L- SERIES
NAM	Name relocatable program.....	yes	yes
NOP	No operation.....	yes	yes
OCT	Define octal constant.....	yes	yes
ORB	Establish origin in base page.....	yes	yes
ORG	Establish program origin.....	yes	yes
ORR	Reset program location counter.....	yes	yes
OTA	Output from A to I/O channel.....	yes	yes
	flag bit.....	yes	yes
OTB	Output from B to I/O channel.....	yes	yes
OTBC	Output from B to I/O channel, clear flag bit.....	yes	yes
PAA*	Load/store Port A map per A.....	yes	N/A
PAB*	Load/store Port A map per B.....	yes	N/A
PBA*	Load/store Port B map per A.....	yes	N/A
PBB*	Load/store Port B map per B.....	yes	N/A
RAL	Rotate A left 1.....	yes	yes
RAM	Generate executable.....	yes	N/A
RAR	Rotate A right 1.....	yes	yes
RBL	Rotate B left 1.....	yes	yes
RBR	Rotate B right 1.....	yes	yes
RELOC	Specify memory space for ensuing code....	yes	yes
REP	Repeat next statement.....	yes	yes
REPEAT	Repeat ensuing code.....	yes	yes
RPL	Replace instruction with microcode.....	yes	yes
RRL	Rotate A and B left.....	yes	yes
RRR	Rotate A and B right.....	yes	yes
RSA	Read status register in A.....	yes	N/A
RSB	Read status register in B.....	yes	N/A
RSS	Reverse skip sense.....	yes	yes
RVA*	Read violation register into A.....	yes	N/A
RVB*	Read violation register into B.....	yes	N/A
SAE	Copy sign bit of A into E.....	yes	yes
SAX	Store A into memory indexed by X.....	yes	*
SAY	Store A into memory indexed by Y.....	yes	*
SBE	Copy sign bit of B into E.....	yes	yes
SBS	Set bits.....	yes	*
SBT	Store byte.....	yes	*
SBX	Store B into memory indexed by X.....	yes	*
SBY	Store B into memory indexed by Y.....	yes	*
SEZ	Skip if E = 0.....	yes	yes
SFB	Scan for byte.....	yes	*

\* Not available in A-Series machines.

## Instructions Index

INSTRUCTION	M/E/F/A- SERIES	L- SERIES
SFC	Skip if I/O flag = 0.....	yes yes
SFS	Skip if I/O flag = 1.....	yes yes
SJP	Enable system map and jump.....	yes N/A
SJS	Enable system map and jump to subroutine.	yes N/A
SKP	Skip to top of next page.....	yes yes
SLA	Skip if lsb of A = 0.....	yes yes
SLB	Skip if lsb of B = 0.....	yes yes
SOC	Skip if overflow bit = 0.....	yes yes
SOCC	Skip if overflow bit = 0, clear flag bit.	yes yes
SOS	Skip if overflow bit = 1.....	yes yes
SOSC	Skip if overflow bit = 1, clear flag bit.	yes yes
SPC	Space n lines.....	yes yes
SSA	Skip if sign A = 0.....	yes yes
SSB	Skip if sign B = 0.....	yes yes
SSM	Store status register in memory.....	yes N/A
STA	Store A.....	yes yes
STB	Store B.....	yes yes
STC	Set I/O control bit.....	yes yes
STCC	Set I/O control bit, clear flag bit.....	yes yes
STF	Set I/O flag.....	yes yes
STO	Set overflow bit.....	yes yes
STX	Store X into memory.....	yes *
STY	Store Y into memory.....	yes *
SUBHEAD	Specify a listing subheading.....	yes yes
SUP	Suppress list output of additional code lines.....	yes yes
SWP	Switch A and B.....	yes yes
SYA	Load/store system map per A.....	yes N/A
SYB	Load/store system map per B.....	yes N/A
SZA	Skip if A = 0.....	yes yes
SZB	Skip if B = 0.....	yes yes
TBS	Test bits.....	yes *
UJP	Enable user map and jump.....	yes N/A
UJS	Enable user map and jump to subroutine...	yes N/A
UNL	Suppress list output.....	yes yes
UNS	Resume list output.....	yes yes
USA	Load/store user map per A.....	yes N/A
USB	Load/store user map per B.....	yes N/A

## Instructions Index

INSTRUCTION	M/E/F/A- SERIES	L- SERIES
XAX	Exchange A and X.....	yes *
XAY	Exchange A and Y.....	yes *
XBX	Exchange B and X.....	yes *
XBY	Exchange B and Y.....	yes *
XCA	Cross compare A.....	yes *
XCB	Cross compare B.....	yes *
XIF	Terminate IFN or IFZ instructions.....	yes yes
XLA	Cross load A.....	yes *
XLB	Cross load B.....	yes *
XMA	Transfer maps internally per A.....	yes N/A
XMB	Transfer maps internally per B.....	yes N/A
XMM	Transfer map or memory.....	yes N/A
XMS	Transfer maps sequentially.....	yes N/A
XOR	Exclusive "or" to A.....	yes yes
XSA	Cross store A.....	yes *
XSB	Cross store B.....	yes *



# Appendix D

## Binary Codes

Table D-1 presents the binary codes for the base set instructions while Table D-2 presents those for the extended instruction group.

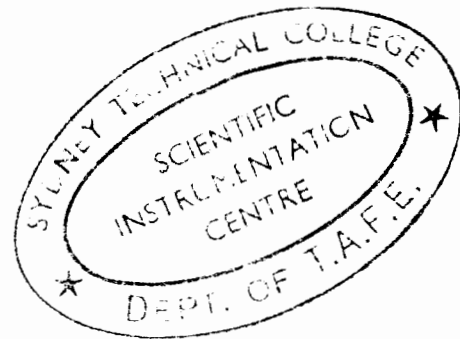


Table D-1. Base Set Instruction Codes in Binary

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/I	AND	001	0	Z/C	← Memory Address →										
D/I	XOR	010	0	Z/C											
D/I	IOR	011	0	Z/C											
D/I	JSB	001	1	Z/C											
D/I	JMP	010	1	Z/C											
D/I	ISZ	011	1	Z/C											
D/I	AD*	100	A/B	Z/C											
D/I	CP*	101	A/B	Z/C											
D/I	LD*	110	A/B	Z/C											
D/I	ST*	111	A/B	Z/C											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	SRG	000	A/B	0	D/E	*LS	000	†CLE	D/E	‡SL*	*LS	000			
			A/B	0	D/E	*RS	001		D/E		*RS	001			
			A/B	0	D/E	R*L	010		D/E		R*L	010			
			A/B	0	D/E	R*R	011		D/E		R*R	011			
			A/B	0	D/E	*LR	100		D/E		*LR	100			
			A/B	0	D/E	ER*	101		D/E		ER*	101			
			A/B	0	D/E	EL*	110		D/E		EL*	110			
			A/B	0	D/E	*LF	111		D/E		*LF	111			
			A/B	0	D/E	L*E	101		D/E		L*E	000			
			A/B	0	D/E	S*E	110		D/E		S*E	000			
			NOP	000	D/E		000		000			000			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	ASG	000	A/B	1	CL*	01	CLE	01	SEZ	SS*	SL*	IN*	SZ*	RSS	
			A/B		CM*	10	CME	10							
			A/B		CC*	11	CCE	11							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	IOG	000		1	H/C	HLT	000	← Select Code →							
				1	0	STF	001								
				1	1	CLF	001								
				1	0	SFC	010								
				1	0	SFS	011								
			A/B	1	H/C	MI*	100								
			A/B	1	H/C	LI*	101								
			A/B	1	H/C	OT*	110								
			0	1	H/C	STC	111								
			1	1	H/C	CLC	111								
				1	0	STO	001			000			001		
				1	1	CLO	001			000			001		
				1	H/C	SOC	010			000			001		
				1	H/C	SOS	011			000			001		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	EAG	000	MPY**	000	010					000			000		
			DIV**	000	100					000			000		
			DLD**	100	010					000			000		
			DST**	100	100					000			000		
			ASR	001	000					0	1				
			ASL	000	000					0	1				
			LSR	001	000					1	0				
			LSL	000	000					1	0				
			RRR	001	001					0	0				
			RRL	000	001					0	0				
Notes:											†CLE: Only this bit is required.				
D/I, A/B, Z/C, D/E, H/C coded: 0/1.											‡SL*: Only this bit and bit 11 (A/B as applicable) are required.				
**Second word is Memory Address.															

Table D-2. Extended Instruction Group Codes in Binary

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SAX/SAY/SBX/SBY	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	0	0	0
CAX/CAY/CBX/CBY	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	0	0	1
LAX/LAY/LBX/LBY	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	0	1	0
STX/STY	1	0	0	0	1	0	1	1	1	1	1	0	X/Y	0	1	1
CXA/CYA/CXB/CYB	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	1	0	0
LDX/LDY	1	0	0	0	1	0	1	1	1	1	1	0	X/Y	1	0	1
ADX/ADY	1	0	0	0	1	0	1	1	1	1	1	0	X/Y	1	1	0
XAX/XAY/XBX/XBY	1	0	0	0	A/B	0	1	1	1	1	1	0	X/Y	1	1	1
ISX/ISY/DSX/DSY	1	0	0	0	1	0	1	1	1	1	1	1	X/Y	0	0	I/D
JUMP INSTRUCTIONS	1	0	0	0	1	0	1	1	1	1	1	1		0	1	0
														JLY = 0		
														JPY = 1		
BYTE INSTRUCTIONS	1	0	0	0	1	0	1	1	1	1	1	0				
														LBT = 0	1	1
														SBT = 1	0	0
														MBT = 1	0	1
														CBT = 1	1	0
														SFB = 1	1	1
BIT INSTRUCTIONS	1	0	0	0	1	0	1	1	1	1	1	1				
														SBS = 0	1	1
														CBS = 1	0	0
														TBS = 1	0	1
WORD INSTRUCTIONS	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	
																CMW = 0
																MVW = 1



Table D-2. Extended Instruction Group Codes in Binary (Continued)






MEMORY EXPANSION	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
DJP/DJS	1	0	0	0	1	0	1	1	1	1	0	1	1						
														DJP = 0 1 0	DJS = 0 1 1				
SYB/USB/PAB/PBB/SSM/JRS	1	0	0	0	1	0	1	1	1	1	0	0	1						
														SYB = 0 0 0	USB = 0 0 1	PAB = 0 1 0	PBB = 0 1 1	SSM = 1 0 0	JRS = 1 0 1
XMA/XLA/XSA/XCA/LFA	1	0	0	0	0	0	1	1	1	1	0	1	0						
														XMA = 0 1 0	XLA = 1 0 0	XSA = 1 0 1	XCA = 1 1 0	LFA = 1 1 1	
MBI/MBF/MBW/MWI/MWF/MWW	1	0	0	0	1	0	1	1	1	1	0	0	0						
														MBI = 0 1 0	MBF = 0 1 1	MBW = 1 0 0	MWI = 1 0 1	MWF = 1 1 0	MWW = 1 1 1
SYA/USA/PAA/PBA	1	0	0	0	0	0	1	1	1	1	0	0	1						
														SYA = 0 0 0	USA = 0 0 1	PAA = 0 1 0	PBA = 0 1 1		

Table D-2. Extended Instruction Group Codes in Binary (Continued)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XMM/XMS/XMB/XLB XSB/XCB/LFB	1	0	0	0	1	0	1	1	1	1	0	1	0			
	<p>XMM = 0 0 0                      XMS = 0 0 1                      XMB = 0 1 0                      XLB = 1 0 0                      XSB = 1 0 1                      XCB = 1 1 0                      LFB = 1 1 1</p>															
RSA/RVA	1	0	0	0	0	0	1	1	1	1	0	1	1			
	<p>RSA = 0 0 0                      RVA = 0 0 1</p>															
RSB/RVB/SJP/SJS/UJP/UJS	1	0	0	0	1	0	1	1	1	1	0	1	1			
	<p>RSB = 0 0 0                      RVB = 0 0 1                      SJP = 1 0 0                      SJS = 1 0 1                      UJP = 1 1 0                      UJS = 1 1 1</p>															



# Appendix E

## Macro Assembler Operations

This Appendix discusses the operations involved with running the Macro/1000 Assembler.

The control statement and assembler options.

- \* The Macro run string.
- \* Messages during assembly.
- \* On-line loading of Macro/1000.



The Macro Assembler is a segmented program that executes under control of RTE in the user program area of main memory. It consists of a main program (MACRO) and 8 segments (MACRO, MACR1, MACR2, MACR3, MACR4, MACR5, MACR6, and MACR7). It resides on disc, and is read into main memory when called by the RU directive.

Source programs, accepted from either an input device, a user file, or Macro on the disc, are translated into absolute or relocatable object programs. MACRO will output the relocatable code, absolute code or macro library to a disc file or device as specified by the destination file parameter when the assembler is invoked.

## Macro Control Statement

The control statement must be the first statement in the source program and it specifies the desired assembler, `MACRO`, and the assembly options:

```
MACRO,p1,p2,p3,....pn
```

The name `MACRO` is in position 1-5 of the statement, followed by a comma and one or more optional parameters, in any order, separated by commas.

The optional parameters, which can be overridden in the Macro run string are:

- A - Absolute Assembly. The addresses generated by the Macro Assembler are to be interpreted as absolute locations in memory. The program is a complete entity; external symbols, common storage references, and entry points are not permitted. Note that an absolute program cannot be executed on RTE (note 1).
- R - Relocatable Assembly. The object program can be loaded anywhere in memory. All operands that refer to memory locations are automatically adjusted as the program is loaded. Programs can contain external symbols and entry points, and can refer to common storage (note 1).
- M - Macro Library Creation. Create macro library and put into the destination file specified in the run string. The destination file must be a Type 1 file (note 2).
- L - List Output. A program listing is to be output to the list file or list device. Listing will contain the object code of the instructions. This includes both the opcode and the address of the operand if it is a memory reference instruction. There are three options:
  - =S short listing, no macro expansion, no conditional assembly.
  - =M medium listing, expand macros, no conditional assembly.
  - =L long listing, expand macros and conditional assembly (notes 3 and 4).

## Macro Assembler Operations

- Q - List Output. A program listing is to be output to the list file or list device. The listing will contain only the operand address for single-word memory reference instructions. In other words, the listing for memory reference instructions contains only the unrelocated address (and not the opcode). The entire instruction will be listed otherwise. The list options used in the L parameter above (=S, =M, =L) can be applied also to the Q parameter (notes 3 and 4).
- T - Symbol Table Print. A listing of the symbol table is to be printed in the listing.
- C - Cross-Reference Table Print. All references to statement labels, external symbols, and user defined opcodes are to be put into the listing along with the source file line numbers at which they are defined and referenced.
- I - Generate Microcode Instructions. The MACRO code-generating feature that generates JSB's for all microcode instructions is overridden. For instance, without this flag, MACRO would generate code for a "JSB .LBT" when the user used the "LBT" opcode. The I option is a command to MACRO to always generate the microcode call for such instructions. Chapter 3 outlines which instructions this affects.
- N,Z - Selective Assemble. These options are included for backward compatibility with old ASMB code. A description of their use is in Appendix L.
- D - Ignored If Specified. This option is for future reference. MACRO will accept this option code, but will take no action on it.
- P - Ignored If Specified. These options are included for backward compatibility with old ASMB code. MACRO will accept these option codes, but will take no action on them.
- X
- Note 1. Since they contradict one another, A and R must never appear in the control statement for the same source program. If neither A nor R is specified, R is assumed.
- Note 2. Only the L, Q and T options can be used with the M option.
- Note 3. If L and Q are both specified, the one specified last will be used.

## Macro Assembler Operations

Note 4. The options that can accompany this parameter are used like this:

```
MACRO,R,L=M,T
```

If no option is appended to the L or Q parameter, the default of 'S' is assumed.

## Run String Parameters

The RTE Macro Assembler is initiated by an RU directive in the following form:

```
: [RU],MACRO,source[,list[,destin[,line/page  
                                ,&.RS1=  
                                [,options ,&.RS2= ]]]]  
                                ,WORK=namr
```

### Source

The source specified may be:

- \* The name of a FMGR file. This file name must conform to the format required by the FMGR namr parameter.
- \* An interactive device LU. The assembler will print a right bracket (]) on the device as a prompt. It will then accept input a line at a time and output another prompt until an EOF is entered.
- \* A non-interactive device LU, such as the mag tape unit or cartridge tape unit.

The source input must always be specified.

## Macro Assembler Operations

### List

Choose one of the following:

- \* (minus symbol). If the source file name begins with an ampersand (&), the list file name will consist of the source file name with the ampersand replaced by an apostrophe ('). For example:

&FIL1	source file name
'FIL1	list file name

The list file is always forced to reside on the same cartridge (cartridge reference number) as the source file.

If a FMGR file by this name does not already exist it is created. The created list file is given the same file security code as that of the source if it was specified in the source namr of the run sequence.

- \* A FMGR file name. This file name must conform to the format required by the FMGR namr parameter. The list file is created if it does not exist.
- \* A logical unit number. The list output is directed to that device.
- \* Null character (omitted). The logical unit of the interactive input device is assumed. If subsequent parameters are specified, the comma must be used as a parameter placeholder.



## Destination

Choose one of the following:

- \* (minus symbol). If the source file name begins with an ampersand (&), the destination file name will consist of the source file name with the ampersand replaced by a percent sign (%) for relocatable code, or an exclamation point (!) for absolute code. For example:

```
&FIL1  source file name
%FIL1  relocatable file name
```

```
&FIL2  source file name
!FIL2  absolute file name
```

This file is forced to reside on the same cartridge (cartridge reference number) as the source file.

If a FMGR file by this name does not already exist it is created.

If the M option is specified (macro library), the minus symbol cannot be used.

The created destination file is given the same file security code as that of the source namr if it was specified in the source namr of the run sequence.

- \* A FMGR file name. The file name must conform to the format requirements set up by the FMGR namr parameter.

If the file does not exist, a relocatable, absolute, or macro library file is created.

If the file exists, it must be type 5 for relocatable binary; type 7 for absolute code; or type 1 for a Macro library. If this condition is not met, Macro prints an error message and aborts.

The recommended first character of the binary file is the percent sign (%) for relocatable code and exclamation point (!) for absolute code. Macro does not insist on this convention.

## Macro Assembler Operations

- \* A logical unit number. The binary output is directed to that logical device.
- \* Null character (omitted). Binary output is not produced. if subsequent parameters are specified, a comma must be used as a parameter placeholder.

### Lines/Page

Enter a decimal number which defines the number of lines per page for the list device.

The default for this parameter is 55 lines per page.

### Options

Enter up to six characters to select control function options. No commas are allowed within the option string. These characters are: A,R,L,M,Q,T,C, and I. Their functions are defined in Figure E-1. If specified when the assembler is run, these options replace (override) the options declared in the MACRO control statement.

Another control function option is P, the override option. It may be used only on the run string. If P is specified by itself, Macro will output just the object code (if the binary output parameter has been specified) and the error reports and take no further action. The type of object code is determined by the source program control statement. If the P option is specified with any other option, it is ignored.

## **&.RS1, &.RS2 or Work**

### **&.RS1, or &.RS2**

The initial values of global system assembly-time variables &.RS1 and &.RS2 may be specified here. They are of type character. If special characters are used (space, comma), the string must be surrounded by single quotes (').

### **Work**

Specify the Type 1 FMGR file name to be used as a work or swapping file. The file name must conform to the format requirements set up by the FMGR namr parameter. If the file does not exist, it is created. If no work file is specified, Macro defaults to a Type 1 file of 96 blocks on the same disc cartridge as the source.

If your source file is large, the work file may be larger than 96 blocks. Specifying the work file in the run string allows for the size and cartridge number to be specified.

## Macro Assembler Operations

### EXAMPLES:

1. :RU,MACRO,&PROGA,-,-

Schedules RTE Macro/1000 to assemble the source code in file &PROGA. Listed output is directed to list file 'PROGA and if relocatable, the destination file is directed to binary file %PROGA. The number of lines per page defaults to 55.

2. :RU,MACRO,&FILE,-,-,,&.RS1='1',&.RS2='1'

&FILE looks like this:

```
MACRO,L,T,R
  NAM FILE
  AIF &.RS1=1
    assemble this section of code
  AELSEIF &.RS2=1
    assemble this section of code
  AENDIF
  END FILE
```

&.RS1 and &.RS2 are flags. The decision as to what sections of code are to be assembled is made without editing the file.

3. :RU,MACRO,&FILE2,-,-,,&.RS1='&FILE3'

&FILE2 looks like this:

```
MACRO,L,T,R
  NAM FILE2
  INCLUDE &.RS1
    remainder of file
  END FILE2
```

&.RS1 is the name of a source file. When &FILE2 is assembled, &FILE3 is also assembled.

4. :RU,MACRO,&FIL1:SC:50,'LIST::55,-

Schedules RTE Macro/1000 to assemble source file &FIL1. Listed output is directed to list file 'LIST on cartridge 55. The destination file if relocatable is %FIL1 with a security code of SC on cartridge 50. The number of lines per page defaults to 55.

## Macro Assembler Operations

### 5. :RU,MACRO,&ABCD

Schedules RTE Macro/1000 to assemble source file &ABCD. Listed output defaults to LU 1 (the user's terminal under RTE-IVB). No destination file is generated. The number of lines per page defaults to 55.

### 6. :RU,MACRO,&AAAA,-,-,28,L=L

Schedules RTE Macro/1000 to assemble source file &AAAA. Listed output is directed to list file 'AAAA. If absolute, the destination file is directed to the binary file !AAAA. The number of lines per page is 28. A long listing will be produced because the L=L option has been specified.

### 7. :RU,MACRO,&SFIL,-,-,,TQC

Schedules RTE Macro/1000 to assemble source file &SFIL. Listed output is directed to the list file 'SFIL. If relocatable, the destination file is directed to the file %SFIL. T will cause a symbol table listing to be output to the list file. Q will cause the memory reference instructions in the object code listing to appear as addresses only (the opcode will not appear in the listing). C will cause a cross-reference table to be output to the list file.

```
:RU,MACRO,&INT4,-,-,,WORK=#INT4::50:1:192
```

Schedules RTE Macro/1000 to assemble source file &INT4. Listed output is directed to the list file 'INT4. If relocatable, the destination file is directed to the file %INT4. The work file #INT4 is specified as a Type 1 file, size 192 blocks on disc cartridge 50.

## Messages During Assembly

If an FMP error occurs during the assembly, the assembler will print the following message on the user's terminal:

```
File manager error # <n>    File = <file>
```

where:

<n> is the FMGR error number.

<file> is the file currently being assembled.

The current assembly is aborted.

If the source input file does not exist, this message appears on the user's terminal:

```
File manager error # -6    File = <ffff>
```

where:

<ffff> is the file name specified in the RU string.

and the current assembly stops.

If an error is found in the assembler control statement, the following message is output to the terminal:

```
Illegal option in run string or control statement.
```

and the current assembly stops.

The following message on the user's terminal signifies the end of the assembly:

```
Macro: No errors total
```

## Macro Assembler Operations

At the end of assembly, Macro displays the error status:

```
Error eee in line nn <macro line # mmm> <include file # iii>
Error eee in line nn <macro line # mmm> <include file # iii>
Error eee in line nn <macro line # mmm> <include file # iii>
```

```
Macro: xx errors total
```

where:

- eee is the error number. See Appendix G for the meaning of error codes.
- nn is the line number where the error occurred.
- xx is the error count at the end of assembly.
- mmm is the line number inside of a macro definition where the error occurred. This phrase is printed only if an error occurred inside the macro.
- iii is the file number of the included file. This phrase is printed only if an error occurred inside of the include file. In this case, nn is the line number in the included file.

Macro will return the number of errors that occurred to the program that scheduled Macro as the first return parameter. This parameter may be retrieved using a call to the library subroutine RMPAR.

## On-Line Loading of the Macro Assembler

The following example illustrates the on-line loading of the Macro Assembler in an RTE Operating System. The size of the program should be increased to at least 18 pages, with 28 pages being the recommended size for large background and 32 pages for extended background. The extra space is used by the Macro Assembler for its symbol table.

MACRO must be loaded as a Type 4 program (large background), or as a Type 6 program (extended background, RTE-6/VM only). This load file is suitable for RTE-6/VM, RTE-XL, and RTE-A.1.

Example:

```

:RU,LOADR

/LOADR:    OP,LB
/LOADR:    SZ,28
/LOADR:    RE,%MACRO
/LOADR:    RE,%MACR0
/LOADR:    RE,%MACR1
/LOADR:    RE,%MACR2
/LOADR:    RE,%MACR3
/LOADR:    RE,%MACR4
/LOADR:    RE,%MACR5
/LOADR:    RE,%MACR6
/LOADR:    RE,%MACR7
/LOADR:    EN

```

You must have the file "M.ERR on your system. "M.ERR is a Type 2 file containing error messages for Macro. The record length for file "M.ERR is 64 words.





# Appendix F

## Cross Reference Table Generator

The cross-reference table generator routine processes a Macro Assembler source program and provides a list of all symbols and symbol references used within the program. To cause MACRO to print the cross-reference table, specify the assembly option 'C' on the MACRO control statement. Each symbol that is defined in a source file will be listed in the cross-reference table. An example of a cross-reference listing is on the next page.

The format is:

```
symbol..... sd(i): srl[*] sr2[*] sr3[*] ... srn[*]
```

where:

symbol is a label found in the assembled file.

sd(i) is the statement number in decimal where symbol is defined. If defined in an include file, i is the include file number.

sri[\*] is a statement number where symbol is referenced. The asterisk (\*) means that the reference was volatile, that is the symbol may be altered. Some examples of a volatile reference at a statement are STA symbol or JMP label, where symbol and label are likely to change because of that statement.

The statements:

```
A EQU 0  
B EQU 1
```

are included in every source file that MACRO assembles. Therefore, they will be in the Cross-Reference Table.

If the symbol is not referenced in the file, the message:

```
symbol not referenced
```

is printed after the defining statement number.

Cross-Reference Table Generator

PAGE# 1 Macro/1000 Version .06 4:51 PM WED., 25 MAR., 1981

```

00001          MACRO,L,R,C
00002          NAM OPERM
00003          MACLIB $SUMLB
00004          ;
00005          ;
00006          INCLUDE DATA
00001          ;
00002          ; DATA SECTION
00003          ;
00004I 00000 000007 INIT      OCT 7      ; DEFINE TWO VARIABLES, INIT
00005I 00001          CHANGE    BSS 1      ; AND CHANGE
00006I 00002 000004 NOT.USED  DEC 4      ; NOT.USED IS NOT REFERENCED
00007I 00000          RELOC COMMON ; USE COMMON
00008I 00000          ABC       BSS 10     ; AND DEFINE AN ARRAY
00007          ;
00008 00003          RELOC PROG
00009 00003 000000 OPERM    NOP
00010 00004 060000R        LDA INIT    ; INIT IS REFERENCED HERE
00011 00005 001300        RAR
00012 00006 070001R        STA CHANGE  ; CHANGE IS ALTERED HERE
00013 00007 064015R        LDB =D6     ; USE A LITERAL
00014 00010 024010R        JMP OVER    ; OVER IS A VOLATILE REFERENCE
00015          ;
00016          ;
00017 00011 000000 OVER    NOP          ; OVER IS DEFINED HERE
00018 00012          QUIT          ; CALL MACRO TO CALL EXIT
00019          ; QUIT WILL CALL EXEC AND
00020          ; PASS IT A VALUE, D6.
          00016 000006
          END OPERM
    
```

Macro/1000 Cross-reference

\* - Volatile reference (store, jump, call...)

```

A . . . . .0: Symbol not referenced
ABC . . . . .8(1): Symbol not referenced
B . . . . .0: Symbol not referenced
CHANGE . . . . .5(1): 12*
D6 . . . . .18: 18
EXEC . . . . .18: 18*
INIT . . . . .4(1): 10
NOT.USED . . . . .6(1): Symbol not referenced
OPERM . . . . .9: 21*
OVER . . . . .17: 14*
Macro: No errors total
    
```

# Appendix G HP-Character Set

BITS		COLUMN		0 <sub>00</sub>	0 <sub>01</sub>	0 <sub>10</sub>	0 <sub>11</sub>	1 <sub>00</sub>	1 <sub>01</sub>	1 <sub>10</sub>	1 <sub>11</sub>
b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	ROW	0	1	2	3	4	5	6	7
0	0	0	0	NUL	DLE	SP	0	@	P		p
0	0	0	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	STX	DC2	"	2	B	R	b	r
0	0	1	1	ETX	DC3	#	3	C	S	c	s
0	1	0	0	EOT	DC4	\$	4	D	T	d	t
0	1	0	1	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	ACK	SYN	&	6	F	V	f	v
0	1	1	1	BEL	ETB	'	7	G	W	g	w
1	0	0	0	BS	CAN	(	8	H	X	h	x
1	0	0	1	HT	EM	)	9	I	Y	i	y
1	0	1	0	LF	SUB	*	:	J	Z	j	z
1	0	1	1	VT	ESC	+	;	K	[	k	{
1	1	0	0	FF	FS	,	<	L	\	l	
1	1	0	1	CR	GS	-	=	M	]	m	}
1	1	1	0	SO	RS	.	>	N	^	n	~
1	1	1	1	SI	US	/	?	O	_	o	DEL

32 CONTROL CODES

Upshifted Lower Case

64 CHARACTER SET

96 CHARACTER SET

128 CHARACTER SET

EXAMPLE: The representation for the character "K" (column 4, row 11) is.

	b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>
BINARY	1	0	0	1	0	1	1
OCTAL	1	1	3				

\* Depressing the Control key while typing an upper case letter produces the corresponding control code on most terminals. For example, Control-H is a backspace.

HEWLETT-PACKARD CHARACTER SET FOR COMPUTER SYSTEMS

This table shows HP's implementation of ANS X3.4-1968 (USASCII) and ANS X3.32-1973. Some devices may substitute alternate characters from those shown in this chart (for example, Line Drawing Set or Scandinavian font). Consult the manual for your device.

The left and right byte columns show the octal patterns in a 16 bit word when the character occupies bits 8 to 14 (left byte) or 0 to 6 (right byte) and the rest of the bits are zero. To find the pattern of two characters in the same word, add the two values. For example, "AB" produces the octal pattern 040502. (The parity bits are zero in this chart.)

The octal values 0 through 37 and 177 are control codes. The octal values 40 through 176 are character codes.

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
32	020000	000040	!	Space, Blank
33	020400	000041	"	Exclamation Point
34	021000	000042	#	Quotation Mark
35	021400	000043	\$	Number Sign, Pound Sign
36	022000	000044	%	Dollar Sign
37	022400	000045	&	Percent
38	023000	000046	'	Ampersand, And Sign
39	023400	000047	(	Apostrophe, Acute Accent
40	024000	000050	(	Left (opening) Parenthesis
41	024400	000051	)	Right (closing) Parenthesis
42	025000	000052	*	Asterisk, Star
43	025400	000053	+	Plus
44	026000	000054	,	Comma, Cedilla
45	026400	000055	-	Hyphen, Minus, Dash
46	027000	000056	.	Period, Decimal Point
47	027400	000057	/	Slash, Slant
48	030000	000060	0	} Digits, Numbers
49	030400	000061	1	
50	031000	000062	2	
51	031400	000063	3	
52	032000	000064	4	
53	032400	000065	5	
54	033000	000066	6	
55	033400	000067	7	
56	034000	000070	8	
57	034400	000071	9	
58	035000	000072	:	Colon
59	035400	000073	;	Semicolon
60	036000	000074	<	Less Than
61	036400	000075	=	Equals
62	037000	000076	>	Greater Than
63	037400	000077	?	Question Mark

Decimal Value	Octal Values		Mnemonic	Graphic <sup>1</sup>	Meaning
	Left Byte	Right Byte			
0	000000	000000	NUL	N <sub>0</sub>	Null
1	000400	000001	SOH	N <sub>1</sub>	Start of Heading
2	001000	000002	STX	N <sub>2</sub>	Start of Text
3	001400	000003	EXT	N <sub>3</sub>	End of Text
4	002000	000004	EOT	N <sub>4</sub>	End of Transmission
5	002400	000005	ENQ	N <sub>5</sub>	Enquiry
6	003000	000006	ACK	N <sub>6</sub>	Acknowledge
7	003400	000007	BEL	N <sub>7</sub>	Bell, Attention Signal
8	004000	000010	BS	N <sub>8</sub>	Backspace
9	004400	000011	HT	N <sub>9</sub>	Horizontal Tabulation
10	005000	000012	LF	N <sub>10</sub>	Line Feed
11	005400	000013	VT	N <sub>11</sub>	Vertical Tabulation
12	006000	000014	FF	N <sub>12</sub>	Form Feed
13	006400	000015	CR	N <sub>13</sub>	Carriage Return
14	007000	000016	SO	N <sub>14</sub>	Shift Out } Alternate Character
15	007400	000017	SI	N <sub>15</sub>	Shift In } Set
16	010000	000020	DLE	N <sub>16</sub>	Data Link Escape
17	010400	000021	DC1	N <sub>17</sub>	Device Control 1 (X-ON)
18	011000	000022	DC2	N <sub>18</sub>	Device Control 2 (TAPE)
19	011400	000023	DC3	N <sub>19</sub>	Device Control 3 (X-OFF)
20	012000	000024	DC4	N <sub>20</sub>	Device Control 4 (TAPE)
21	012400	000025	NAK	N <sub>21</sub>	Negative Acknowledge
22	013000	000026	SYN	N <sub>22</sub>	Synchronous Idle
23	013400	000027	ETB	N <sub>23</sub>	End of Transmission Block
24	014000	000030	CAN	N <sub>24</sub>	Cancel
25	014400	000031	EM	N <sub>25</sub>	End of Medium
26	015000	000032	SUB	N <sub>26</sub>	Substitute
27	015400	000033	ESC	N <sub>27</sub>	Escape <sup>2</sup>
28	016000	000034	FS	N <sub>28</sub>	File Separator
29	016400	000035	GS	N <sub>29</sub>	Group Separator
30	017000	000036	RS	N <sub>30</sub>	Record Separator
31	017400	000037	US	N <sub>31</sub>	Unit Separator
127	077400	000177	DEL	N <sub>127</sub>	Delete, Rubout <sup>3</sup>



Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
96	060000	000140	`	Grave Accent <sup>5</sup>
97	060400	000141	a	
98	061000	000142	b	
99	061400	000143	c	
100	062000	000144	d	
101	062400	000145	e	
102	063000	000146	f	
103	063400	000147	g	
104	064000	000150	h	
105	064400	000151	i	
106	065000	000152	j	
107	065400	000153	k	
108	066000	000154	l	
109	066400	000155	m	
110	067000	000156	n	
111	067400	000157	o	
112	070000	000160	p	
113	070400	000161	q	
114	071000	000162	r	
115	071400	000163	s	
116	072000	000164	t	
117	072400	000165	u	
118	073000	000166	v	
119	073400	000167	w	
120	074000	000170	x	
121	074400	000171	y	
122	075000	000172	z	
123	075400	000173	{	
124	076000	000174		
125	076400	000175	}	
126	077000	000176	~	

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
64	040000	000100	@	Commercial At
65	040400	000101	A	
66	041000	000102	B	
67	041400	000103	C	
68	042000	000104	D	
69	042400	000105	E	
70	043000	000106	F	
71	043400	000107	G	
72	044000	000110	H	
73	044400	000111	I	
74	045000	000112	J	
75	045400	000113	K	
76	046000	000114	L	
77	046400	000115	M	
78	047000	000116	N	
79	047400	000117	O	
80	050000	000120	P	
81	050400	000121	Q	
82	051000	000122	R	
83	051400	000123	S	
84	052000	000124	T	
85	052400	000125	U	
86	053000	000126	V	
87	053400	000127	W	
88	054000	000130	X	
89	054400	000131	Y	
90	055000	000132	Z	
91	055400	000133	[	
92	056000	000134	\	
93	056400	000135	]	
94	057000	000136	^	
95	057400	000137	~	

Notes: <sup>1</sup>This is the standard display representation. The software and hardware in your system determine if the control code is displayed, executed, or ignored. Some devices display all control codes as '^', '@', or space.

<sup>2</sup>Escape is the first character of a special control sequence. For example, ESC followed by "J" clears the display on a 2640 terminal.

<sup>3</sup>Delete may be displayed as " \_ ", "@", or space.

<sup>4</sup>Normally, the caret and underline are displayed. Some devices substitute the up arrow and back arrow.

<sup>5</sup>Some devices upshift lower case letters and symbols ( ` through ~ ) to the corresponding upper case character ( @ through ^ ). For example, the left brace would be converted to a left bracket.

## HP 7970B BCD-ASCII CONVERSION

SYMBOL	BCD (OCTAL CODE)	ASCII EQUIVALENT (OCTAL CODE)	SYMBOL	BCD (OCTAL CODE)	ASCII EQUIVALENT (OCTAL CODE)
(space)	20	040	@	14	100
!	52	041	A	61	101
"	37	042	B	62	102
#	13	043	C	63	103
\$	53	044	D	64	104
%	57	045	E	65	105
&	†	046	F	66	106
'	35	047	G	67	107
(	34	050	H	70	110
)	74	051	I	71	111
*	54	052	J	41	112
+	60	053	K	42	113
,	33	054	L	43	114
-	40	055	M	44	115
.	73	056	N	45	116
/	21	057	O	46	117
0	12	060	P	47	120
1	01	061	Q	50	121
2	02	062	R	51	122
3	03	063	S	22	123
4	04	064	T	23	124
5	05	065	U	24	125
6	06	066	V	25	126
7	07	067	W	26	127
8	10	070	X	27	130
9	11	071	Y	30	131
:	15	072	Z	31	132
;	56	073	[	75	133
<	76	074	\	36	134
=	17	075	]	55	135
>	16	076	↑	77	136
?	72	077	←	32	137

†The ASCII code 046 is converted to the BCD code for a space (20) when writing data onto a 7-track tape.

# Appendix H Relocatable Record Formats

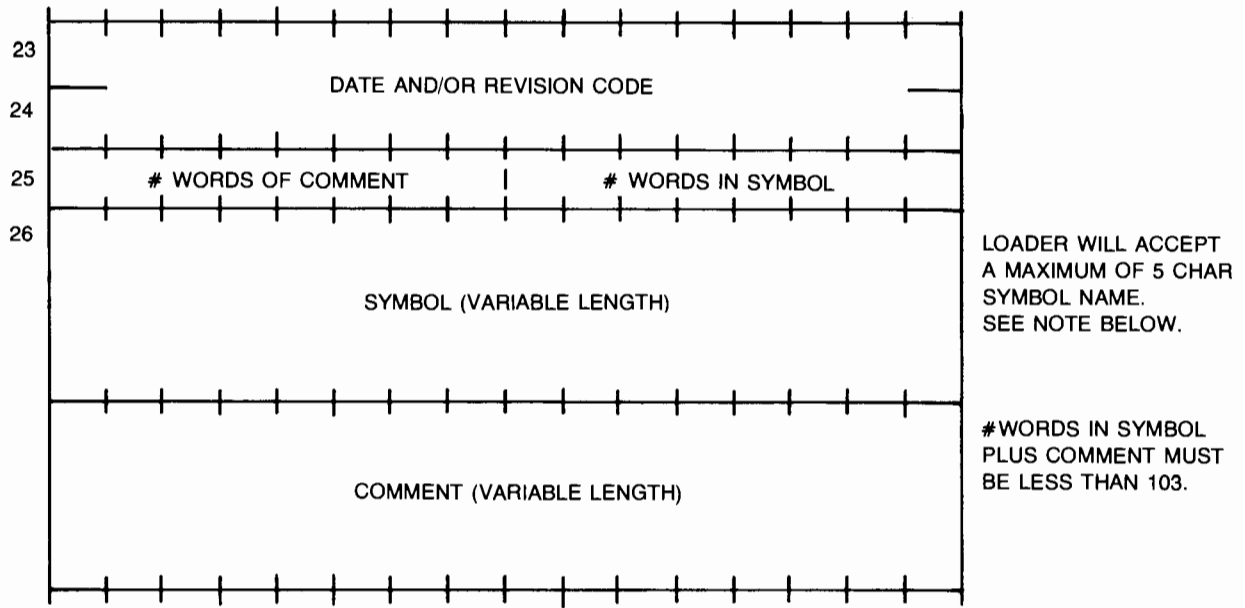
## NOTE

Hewlett-Packard reserves the right to modify the form of these data structures without notice to the customer.

### EXTENDED NAM RECORD

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	EXPLANATION
1	RECORD LENGTH											////////////////////					REC LENGTH <128 WORDS
2	IDENT =7					SUB-IDENT =1					OFFSET OF SIZ WRD					CHECKSUM: ARITHMETIC TOTAL OF ALL WORDS EXCEPT WORDS 1 AND 3	
3	CHECKSUM																
4	LOCAL EMA SIZE																SIZE IS IN WORDS
5	SAVE SIZE																SIZE IS IN WORDS
6	PROGRAM SIZE																SIZE IS IN WORDS
7	0	PROGRAM SIZE															SIZE IS IN WORDS
8	BASE PAGE SIZE																SIZE IS IN WORDS
9	COMMON SIZE																SIZE IS IN WORDS
10	PROGRAM TYPE																
11	PRIORITY																
12	RESOLUTION CODE																
13	EXECUTION MULTIPLE																
14	HOURS																
15	MINUTES																
16	SECONDS																
17	10'S OF MILLISECONDS																
18	<RESERVED>																
19	YEAR OF COMPILATION																
20	JULIAN DAY								HOUR								
21	MINUTES								SECONDS								
22	<RESERVED>																





///// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

NOTE

All Operating systems allow space for only five characters in main-program and segment names.

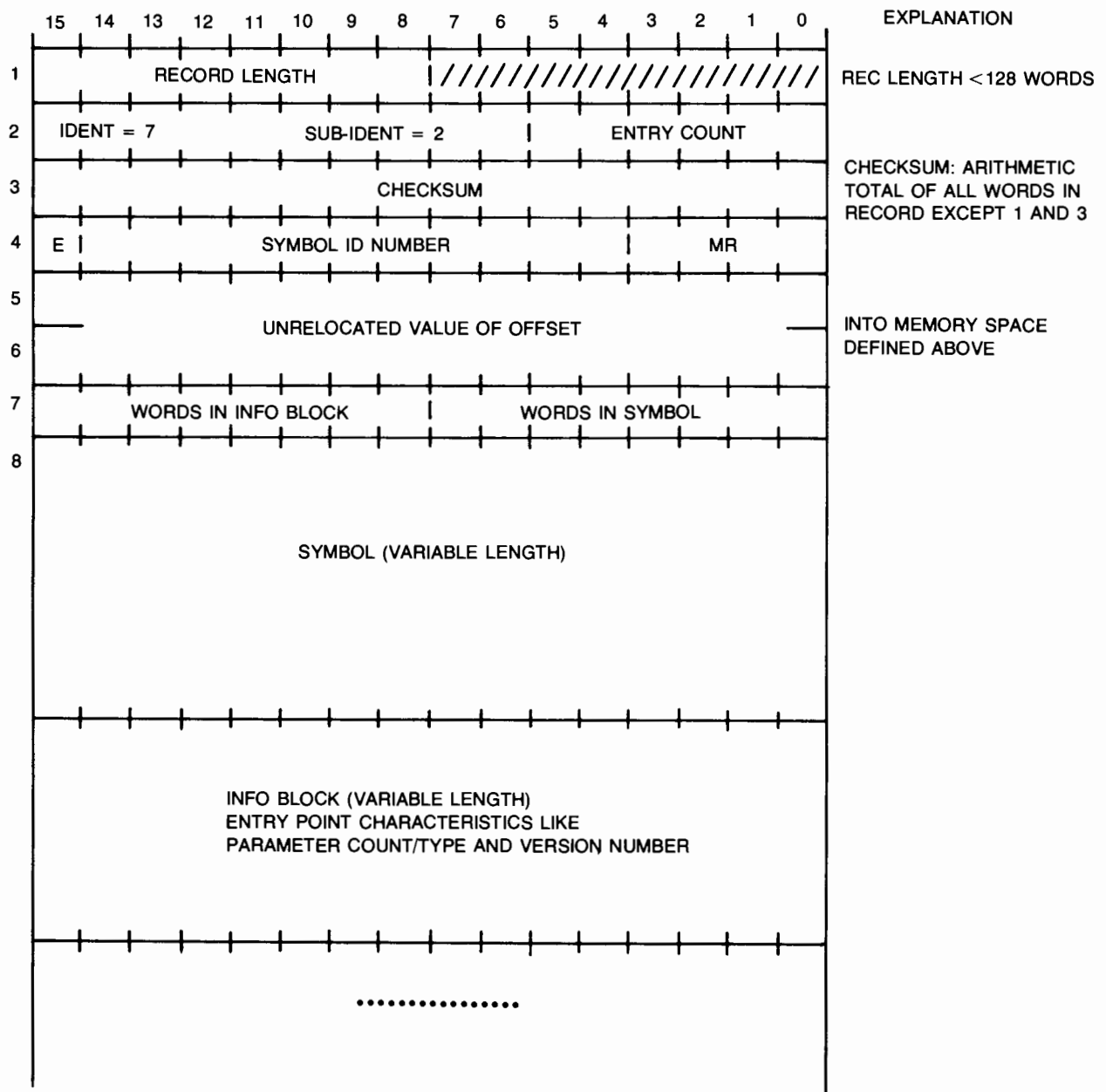
The loaders for RTE-6/VM and RTE-XL will handle symbols longer than five characters.

The loaders for RTE-4B, RTE-4E and earlier operating systems will not handle the longer symbols.

All system generators are limited to five-character symbols, module names and entry points.

Use of the OLDRE Utility is recommended to ensure conformance to these standards.

**EXTENDED ENT RECORD**

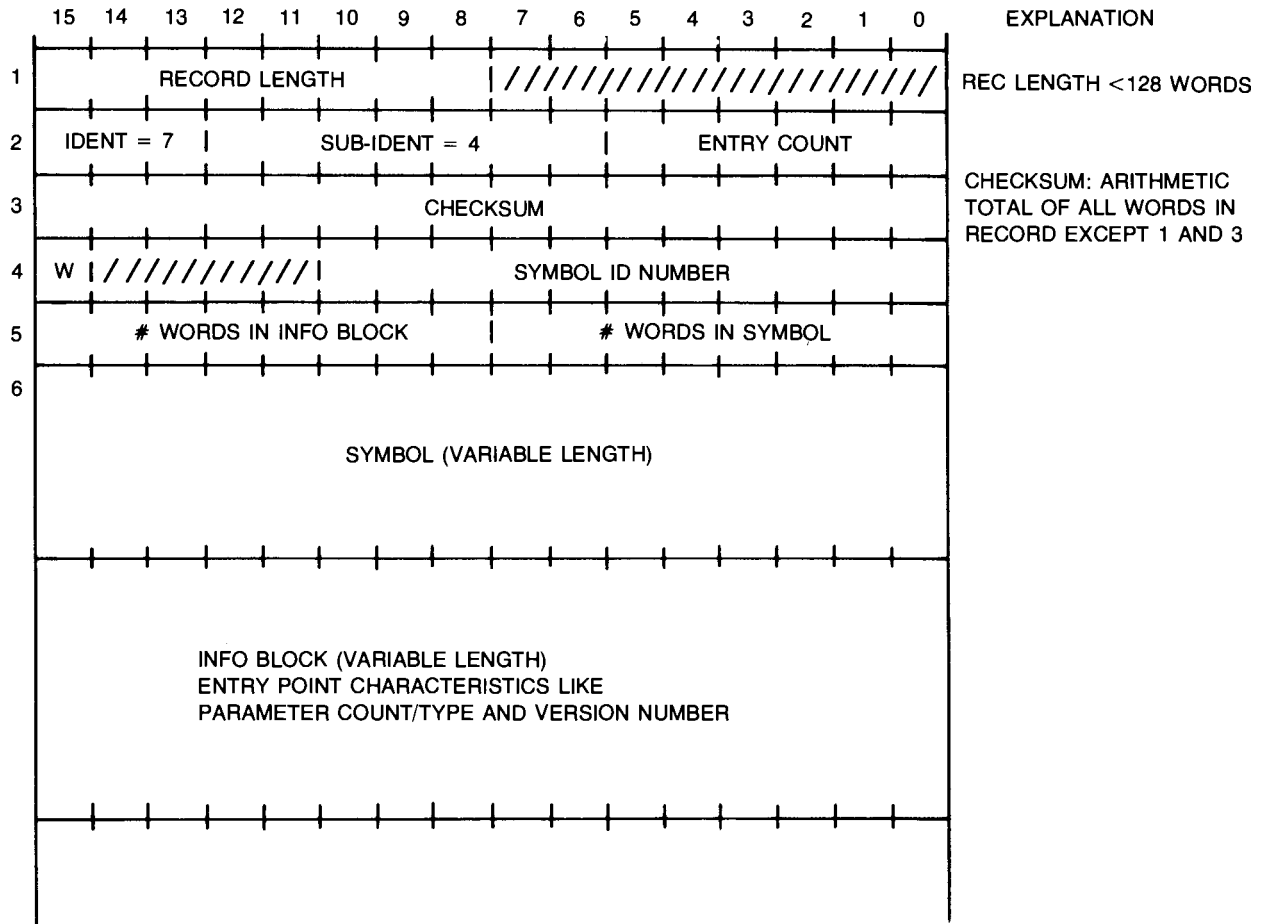


////// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

E = 1, IF EMA  
= 0, IF MR

MR IS MEMORY SPACE  
= 0, IF ABSOLUTE SPACE  
= 1, IF PROGRAM RELOCATABLE SPACE  
= 2, IF BASE PAGE RELOCATABLE SPACE  
= 3, IF COMMON RELOCATABLE SPACE  
= 4, <RESERVED>  
= 5, IF EMA SPACE  
= 6, IF SAVE AREA SPACE

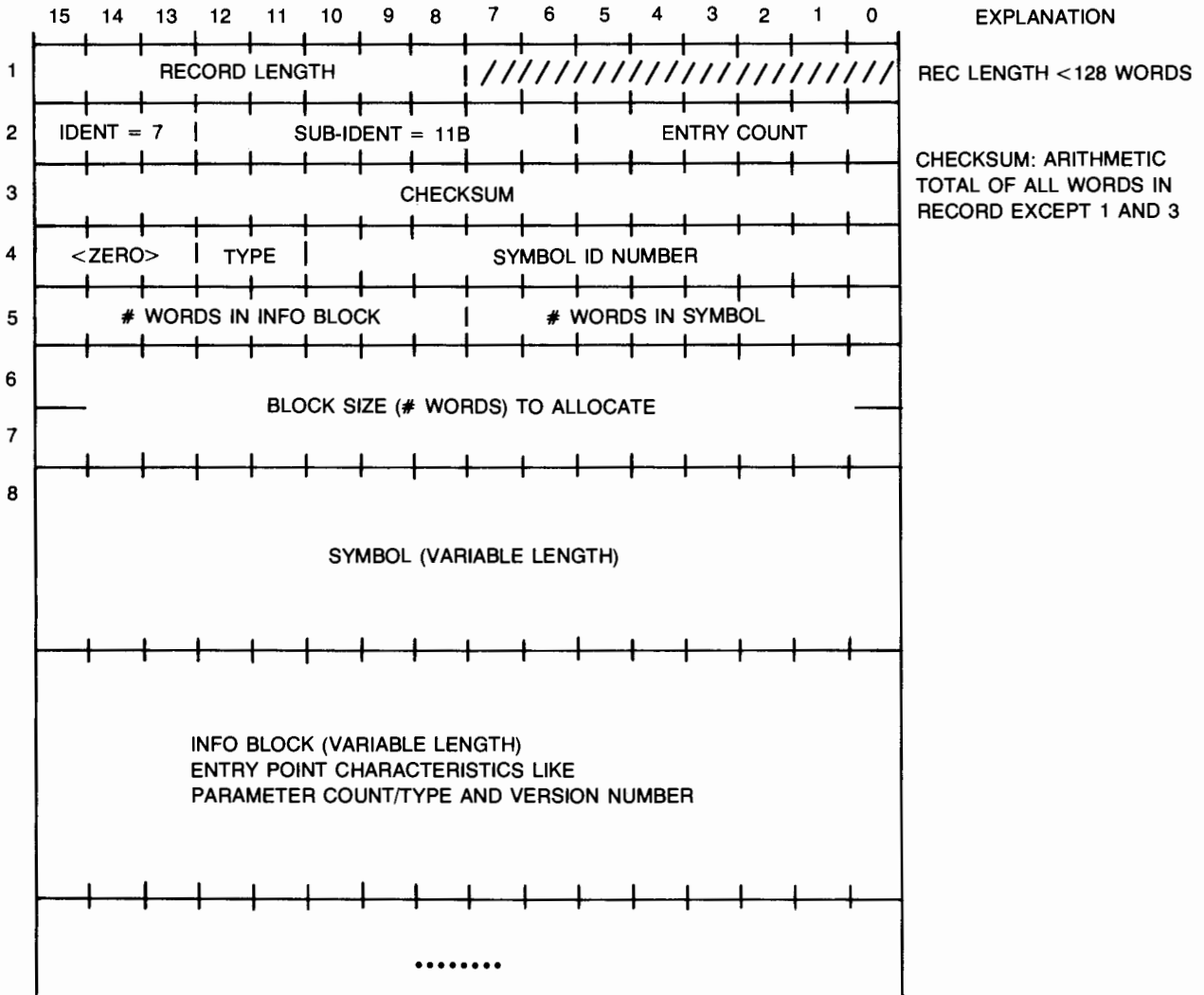
**EXTENDED EXT RECORD**



///// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

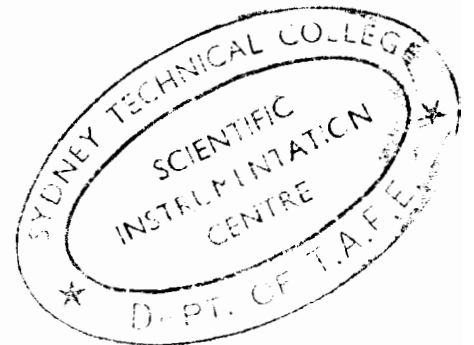
W = 1, IF WEAK EXTERNAL  
= 0, IF REGULAR EXTERNAL

**ALLOCATE RECORD**

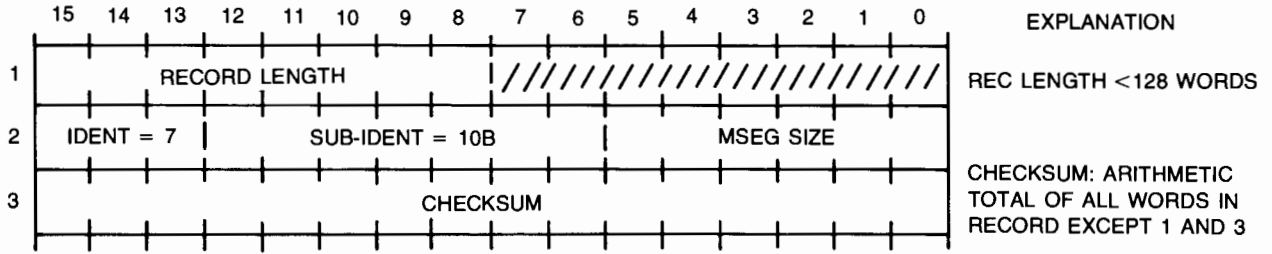


////// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

- TYPE = 0, IF NAMED COMMON (PROGRAM ALLOCATE)  
 1, IF NAMED SAVE COMMON (SAVE ALLOCATE)  
 2, IF NAMED EMA COMMON (EMA ALLOCATE)



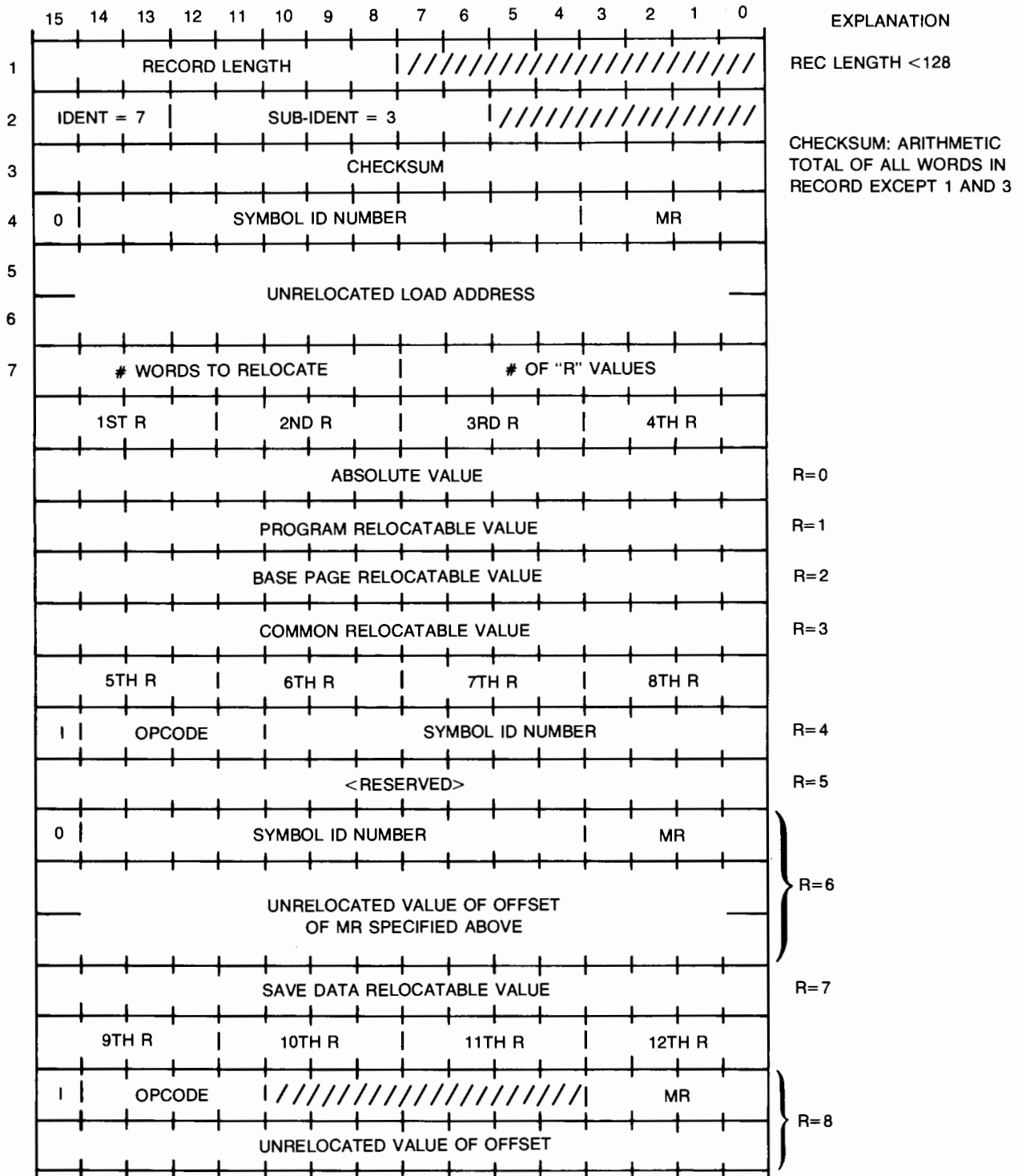
**MSEG RECORD**



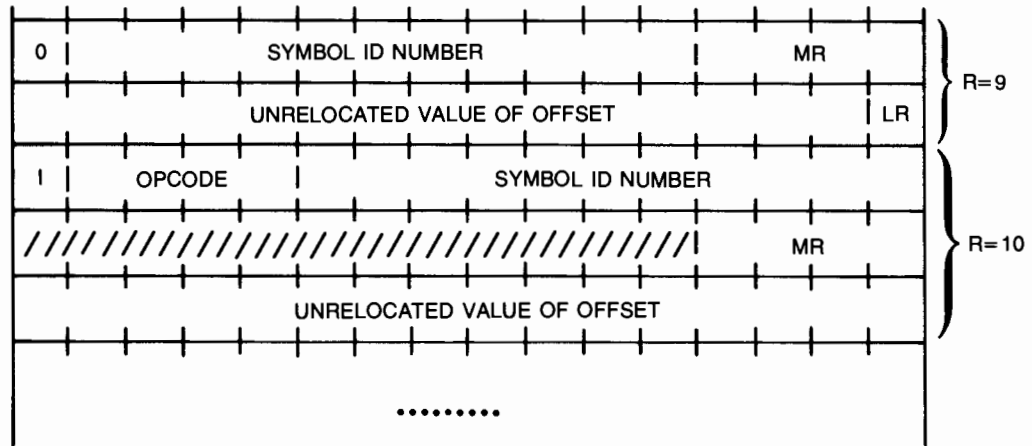
////// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

MSEG SIZE IN PAGES.  $1 \leq \text{MSEG SIZE} \leq 32$

**EXTENDED DBL RECORD**



**EXTENDED DBL RECORD (continued)**

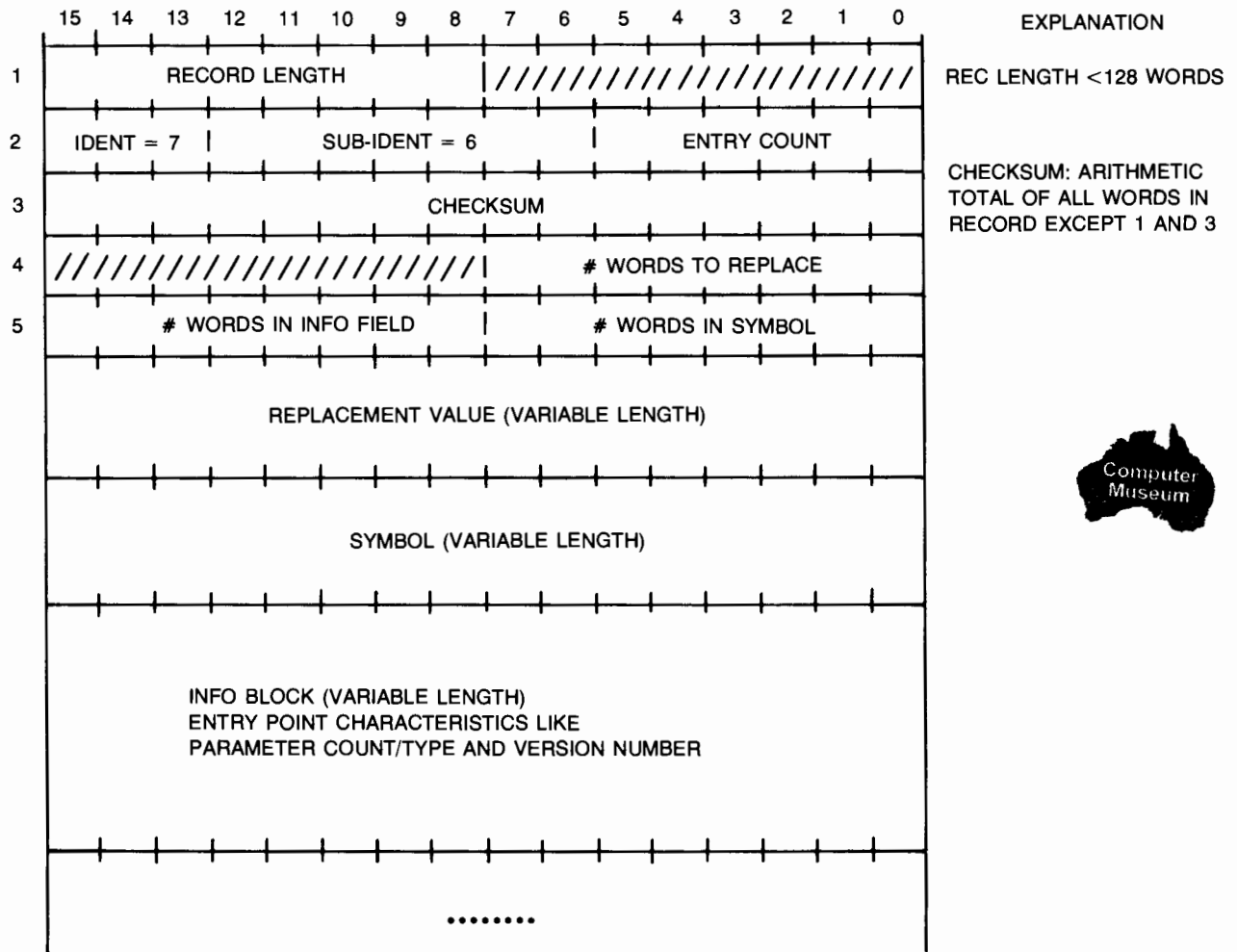


//////// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

**MR IS MEMORY SPACE**

- = 0, IF ABSOLUTE SPACE
- = 1, IF PROGRAM RELOCATABLE SPACE
- = 2, IF BASE PAGE RELOCATABLE SPACE
- = 3, IF COMMON RELOCATABLE SPACE
- = 4, <RESERVED>
- = 5, IF EMA SPACE
- = 6, IF SAVE AREA SPACE

**RPL RECORD**

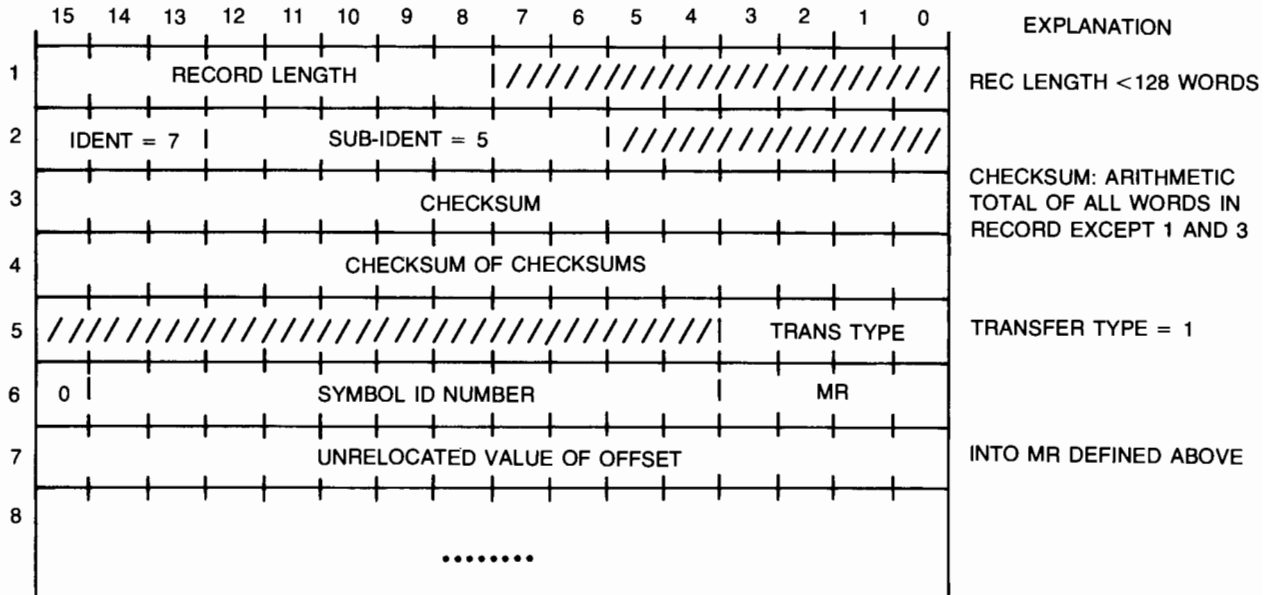


///// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.





**EXTENDED END RECORD**

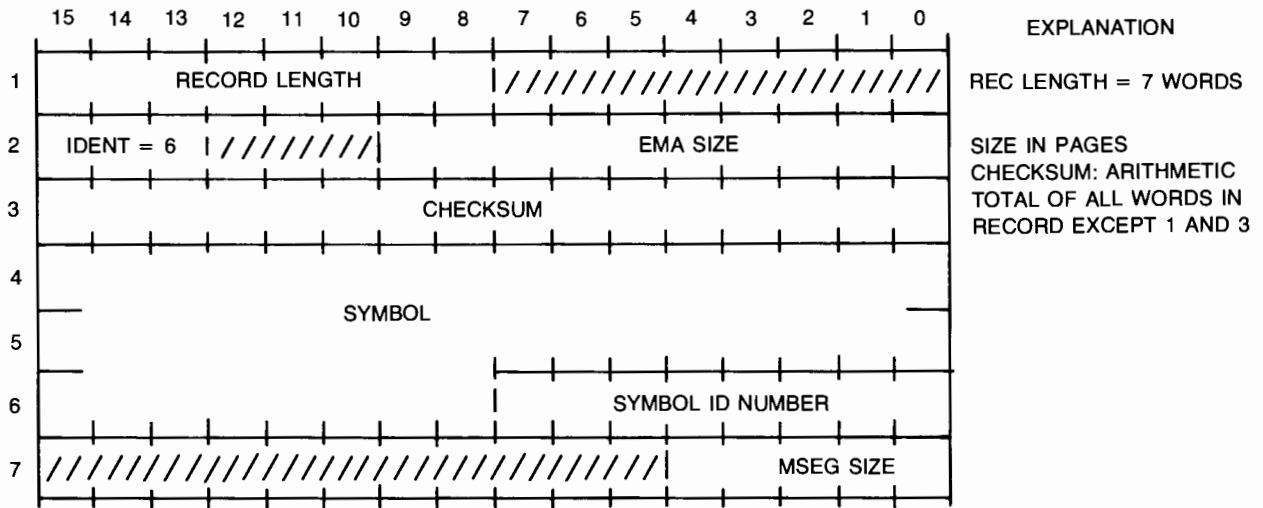


///// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

**MR IS MEMORY SPACE**

- = 0, IF ABSOLUTE SPACE
- = 1, IF PROGRAM RELOCATABLE SPACE
- = 2, IF BASE PAGE RELOCATABLE SPACE
- = 3, IF COMMON RELOCATABLE SPACE
- = 4, <RESERVED>
- = 5, IF EMA SPACE
- = 6, IF SAVE AREA SPACE

**EMA RECORD**

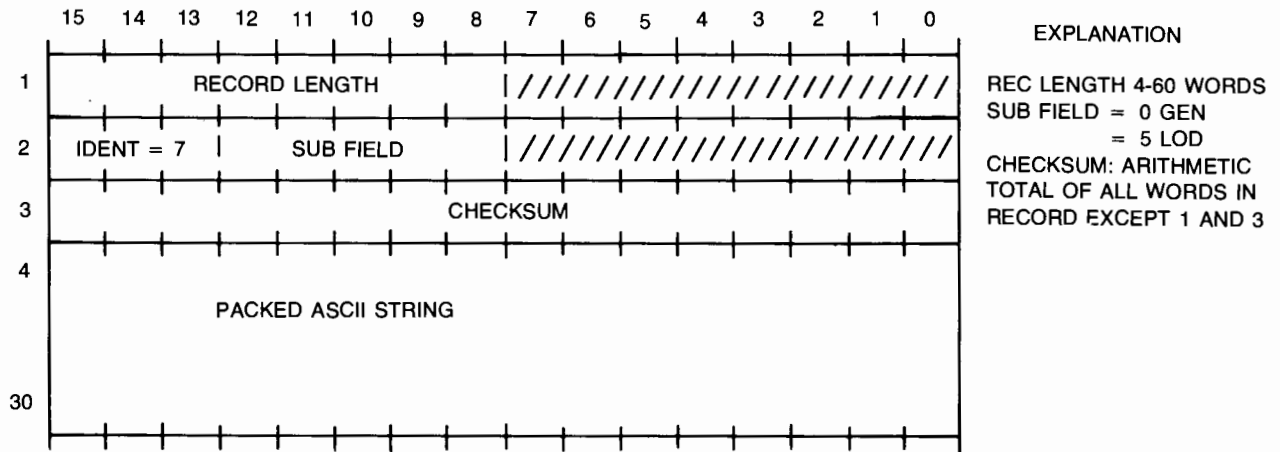


**EXPLANATION**  
 REC LENGTH = 7 WORDS  
 SIZE IN PAGES  
 CHECKSUM: ARITHMETIC  
 TOTAL OF ALL WORDS IN  
 RECORD EXCEPT 1 AND 3

///// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

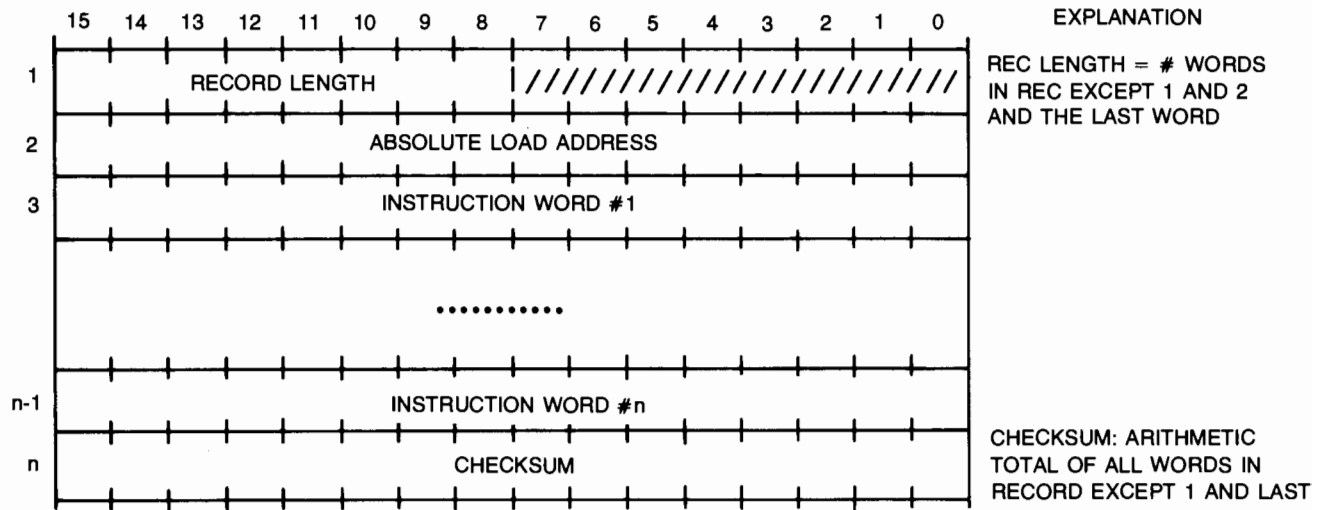
MSEG SIZE IS IN PAGES, 1 ≤ MSEG SIZE ≤ 32

**LOADER/GENERATOR INFORMATION RECORD**



**EXPLANATION**  
 REC LENGTH 4-60 WORDS  
 SUB FIELD = 0 GEN  
 = 5 LOD  
 CHECKSUM: ARITHMETIC  
 TOTAL OF ALL WORDS IN  
 RECORD EXCEPT 1 AND 3

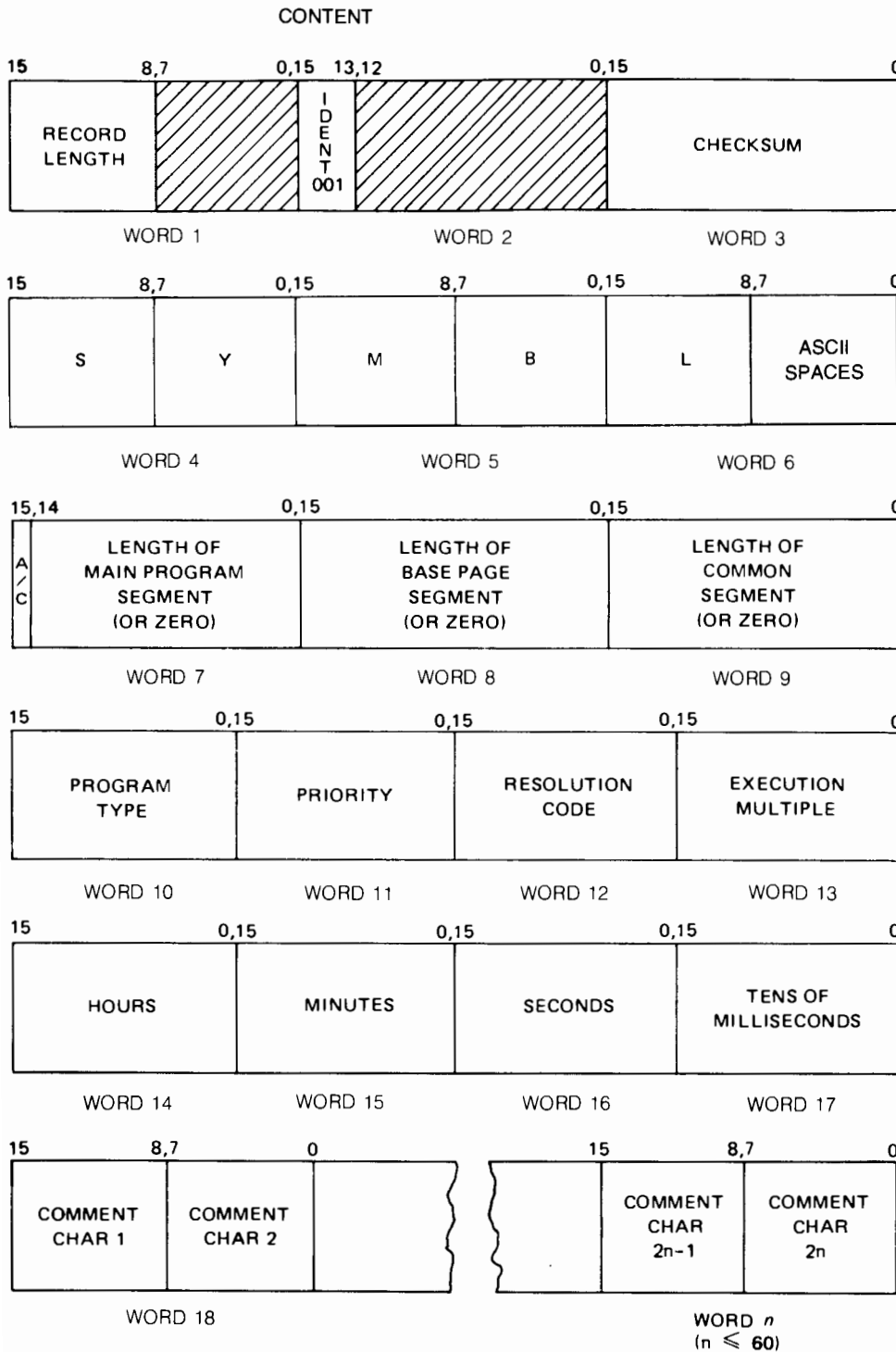
**ABSOLUTE FORMAT**



ABSOLUTE LOAD ADDRESS: STARTING ADDRESS FOR LOADING THE INSTRUCTIONS WHICH FOLLOW.

INSTRUCTION WORDS: ABSOLUTE INSTRUCTIONS OR DATA.

**NAM RECORD**



EXPLANATION

RECORD LENGTH = 9-60 WORDS

IDENT = 001

CHECKSUM: ARITHMETIC TOTAL OF ALL WORDS IN RECORD EXCLUDING WORDS 1 AND 3.

SYMBL: FIVE CHARACTER NAME OF PROGRAM

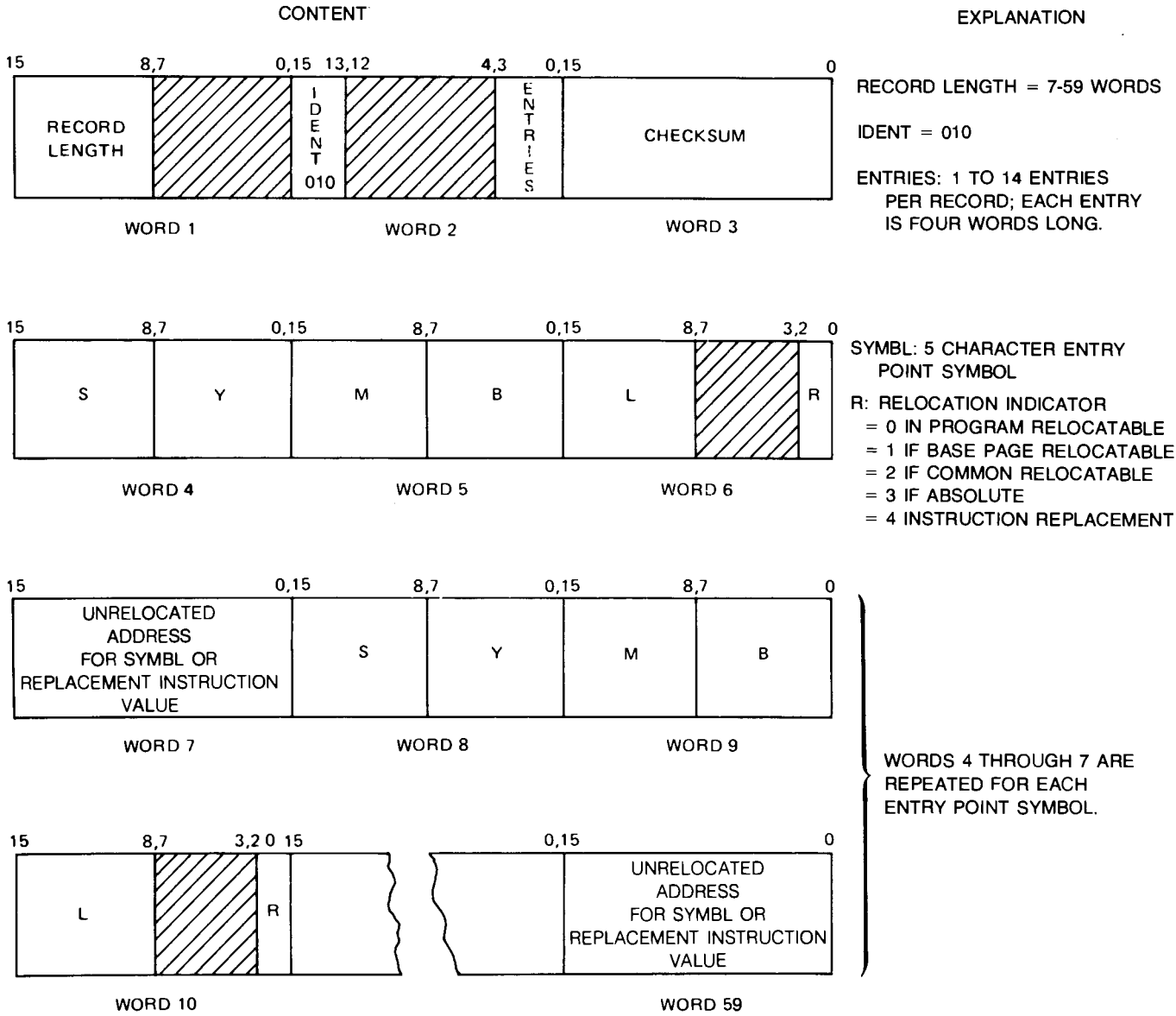
A/C: BINARY TAPE PRECESSION

= 0 IF ASSEMBLER PRODUCED OR LENGTH IS EXACT.

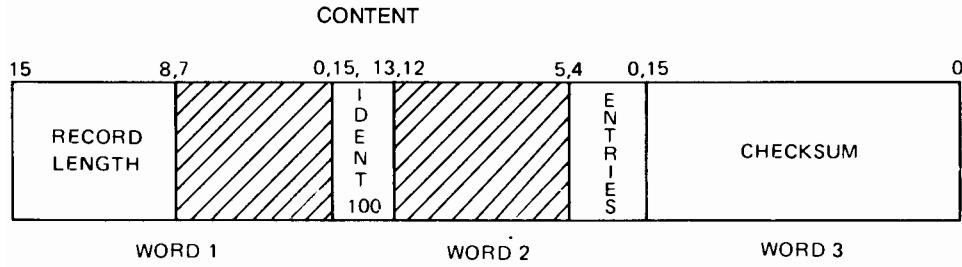
= 1 IF COMPILER PRODUCED AND LENGTH IS UNKNOWN.

HATCH-MARKED AREAS SHOULD BE ZERO-FILLED WHEN THE RECORDS ARE GENERATED

ENT RECORD



**EXT RECORD**

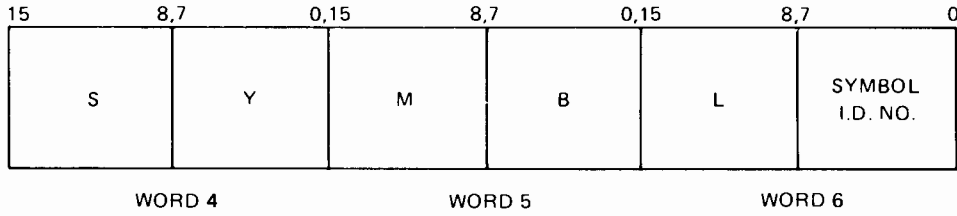


EXPLANATION

RECORD LENGTH = 6-60 WORDS

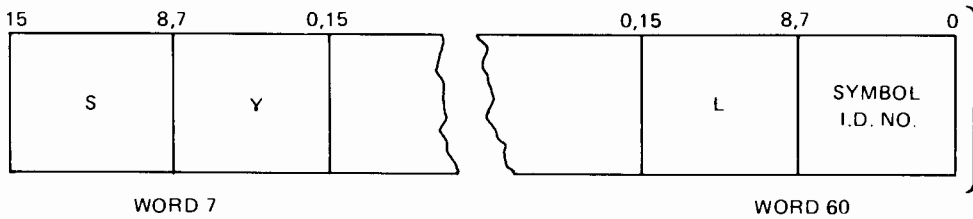
IDENT = 100

ENTRIES: 1 TO 19 PER RECORD; EACH ENTRY IS THREE WORDS LONG



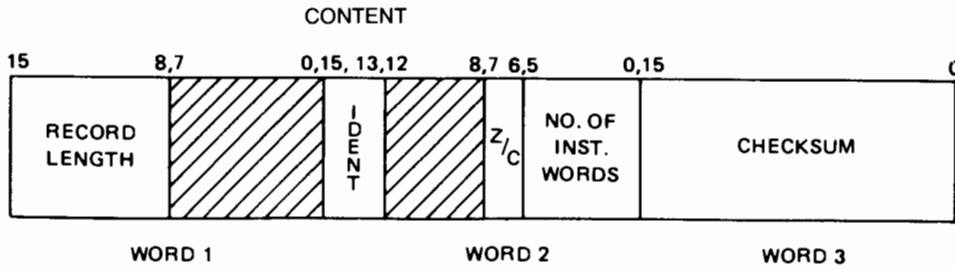
SYMBL: 5 CHARACTER EXTERNAL SYMBOL

SYMBOL ID. NO.: NUMBER ASSIGNED TO SYMBL FOR USE IN LOCATING REFERENCE IN BODY OF PROGRAM.



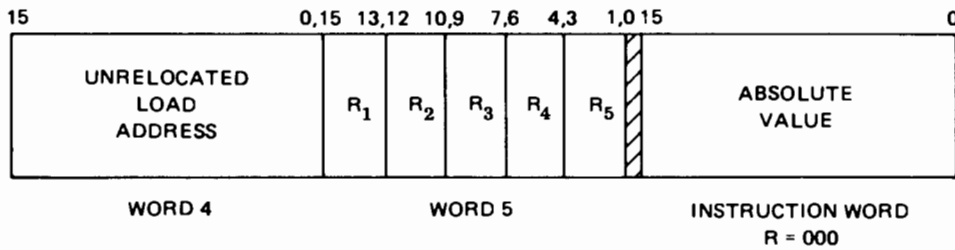
WORDS 4 THROUGH 6 REPEATED FOR EACH EXTERNAL SYMBOL (MAXIMUM OF 19 PER RECORD).

**DBL RECORD**



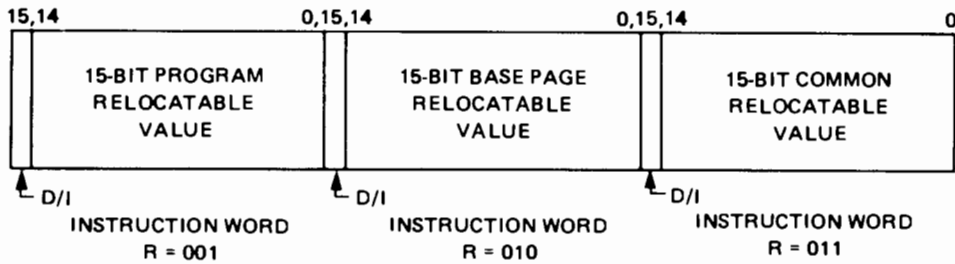
**EXPLANATION**

RECORD LENGTH = 6-60 WORDS  
 IDENT = 011  
 Z/C: RELOCATION OF LOAD ADDRESS  
     = 0 FOR BASE PAGE  
     = 1 FOR PROGRAM  
     = 2 FOR ABSOLUTE  
     = 3 FOR COMMON  
 NO. OF INST. WORDS: 1 TO 45  
 LOADABLE INSTRUCTION WORDS PER RECORD

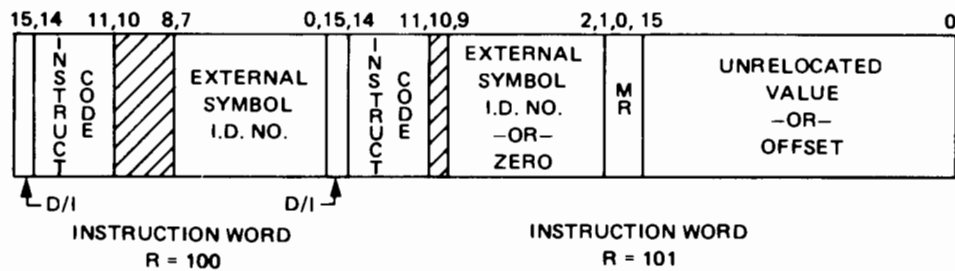


RELOCATABLE LOAD ADDRESS: STARTING ADDRESS FOR LOADING THE INSTRUCTIONS WHICH FOLLOW;

R's: RELOCATION INDICATORS:  
 000 = ABSOLUTE  
 001 = 15-BIT PROGRAM RELOCATABLE  
 010 = 15-BIT BASE PAGE RELOCATABLE  
 011 = 15-BIT COMMON RELOCATABLE  
 100 = EXTERNAL REFERENCE  
 101 = MEMORY REFERENCE  
 110 = BYTE ADDRESS



R<sub>1</sub> IS RELOCATION INDICATOR FOR INSTRUCTION WORD<sub>1</sub>; R<sub>2</sub>, FOR INSTRUCTION WORD<sub>2</sub>; ETC.

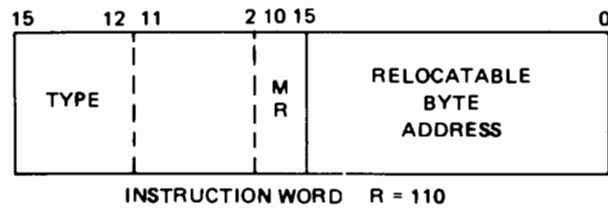


D/I: INDIRECT ADDRESSING

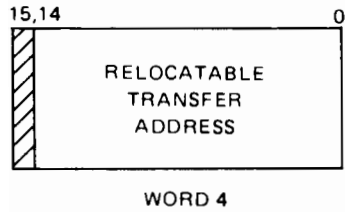
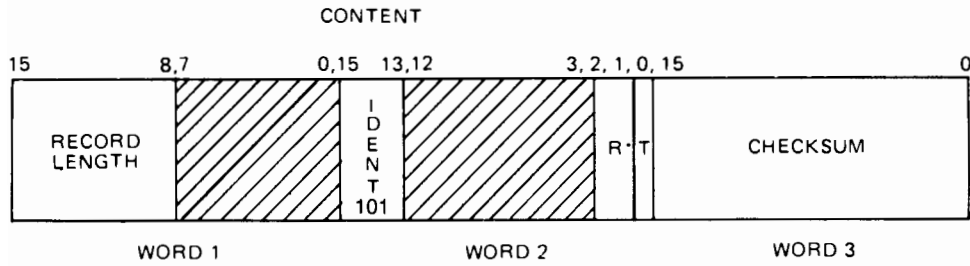
0 = DIRECT  
 1 = INDIRECT

MEMORY REFERENCE INSTRUCTIONS USE TWO WORDS, WITHIN THE TWO-WORD GROUP "MR" INDICATES RELOCATABILITY OF OPERAND SPECIFIED IN SECOND WORDS:

00 = PROGRAM RELOCATABLE  
 01 = BASE PAGE RELOCATABLE  
 10 = COMMON RELOCATABLE  
 11 = ABSOLUTE



END RECORD



EXPLANATION

RECORD LENGTH = 4 WORDS  
IDENT = 101

R: RELOCATION INDICATOR  
FOR TRANSFER ADDRESS

- = 0 IF PROGRAM RELOCATABLE
- = 1 IF BASE PAGE RELOCATABLE
- = 2 IF COMMON RELOCATABLE
- = 3 IF ABSOLUTE

T: TRANSFER ADDRESS  
INDICATOR

- = 0 IF NO TRANSFER  
ADDRESS IN RECORD
- = 1 IF TRANSFER ADDRESS  
PRESENT





# Appendix I

## Implementation Notes

The following tasks are performed on each pass of MACRO.

Pass 1:

1. Macro definition and expansion.
2. Conditional assembly.
3. Assembly-time variable manipulation. (GLOBAL, LOCAL, and SET statements.)
4. String substitution and concatenation.
5. INCLUDE
6. Repetition statements. (REP, REPEAT, AWHILE.)
7. MACLIB
8. Selective assembly (IFN,IFZ) is performed.
9. Prepare intermediate file for Pass 2 containing all of the code that is to be executed, and notation pertaining to what will be listed in the final pass.

Pass 2:

1. User labels entered into symbol table, along with relocatable values.
2. Literals are processed, put into literals table, and space allocated for the literals block. The LIT command is processed.
3. EQUs put into symbol table.
4. Each opcode is examined for its length, so that the program relocation counter can be maintained. ORB, ORG, and RELOC are processed.
5. Selective assembly (IFN, IFZ) is performed.

## Implementation Notes

6. MIC instruction is processed, so the length of any user-defined instruction is known.
7. The file produced on Pass 1 is left for Pass 3 unchanged.
8. The symbol table is preserved for Pass 3.

### Pass 3:

1. Code is generated for each machine opcode.
2. Literals are processed and values placed in literals block.
3. Listing file is produced.
4. Code substitution for the MIC and RAM instruction is performed.

### Pass 4:

This pass is optional, and is specified by the use of the C parameter on the Control Statement.

1. Produce a cross reference table, and append it to the users listing.

There will be one for each of the first 3 passes.

### Pass 5:

This pass is optional, depending on whether the user requires absolute assembly or not.

1. Schedule the absolute assembly post processor to convert the relocatable records produced into an absolute form. This must be done since absolute programs are produced in the same way relocatables are.

# Appendix J

## System Assembly Time Variables

The assembler declares system assembly-time variables (ATVs). Its value is available to you, and in some cases, you can set it. All system ATVs start with ampersand period (&.). the period distinguishes them from other assembly-time variables; therefore, do not use a period as the first character after the & in your own variable names.

Here are the system assembly-time variables:

&.Q	&.RS1	&.REP
&.ERROR	&.RS2	&.PCOUNT

### &.Q

&.Q is a unique number and is type integer. It is local to the macro within which it is used and contains a unique number for each separate macro. In other words, every time a macro is called, &.Q is incremented, thus making it unique from the last time that macro was called.

It can be used in macros that define labels to avoid creating a doubly defined symbol.

Example:

```
MACRO
    TEST  &P1, &P2
    :
    JMP   L&.Q
    :
L&.Q  NOP
    :
    ENDMAC
```

Like all type-integer assembly-time variables, when substitution is done, leading zeroes are suppressed. For instance, the example above could generate a label 'L72' but not 'L0072'.

## **&.ERROR**

**&.ERROR** contains the assembly error count and is type-integer global. Everytime an assembler error is detected, **&.ERROR** is incremented by one.

Like any assembly-time variable, its value can be changed with an ISET statement. It can be used to count user-defined errors. Using conditional assembly, some condition could be tested and, if true, **&.ERROR** could be incremented.

Example:

```
MACRO
    COPY  &P1
    AIF   16>=&P1
&.ERROR ISET  &.ERROR+1
*<<< parameter to macro COPY must be < 16 characters
    AENDIF
    :
    ENDMAC
```

Notice the comment after dumping the error count. This comment line will be output to the list file if LIST LONG is specified.

If **&.ERROR** is changed while assembling is taking place, the line:

```
User-defined errors detected
```

is listed at the end of the list file.

The pseudo op MNOTE will increment **&.ERROR** automatically and print out an ASCII string in the program listing. See the description of MNOTE in Chapter 4.

## &.RS1 and &.RS2

&.RS1 and &.RS2 are type-character global assembly-time variables. They can be used as optional parameters in the macro run string as follows:

```
:RU,MACRO,&PROG,-,-,,,&.RS1='A',
```

and then in the program, they can be tested:

```
AIF  &.RS1='A'  
:  
AELSE  
:  
AENDIF
```

If no values are entered, &.RS1 and &.RS2 are initialized to a string length of zero. These variables can be changed by means of the CSET statement.

## &.REP

&.REP is a type-integer ATV that is local to an AWHILE or REPEAT loop. It is a count of the number of times the loop has been repeated.

Example:

```
&SEE      IGLOBAL      0  
          AWHILE      &SEE < 5  
          :  
&SEE      ISET         &.REP  
          AENDWHILE
```

## **&.PCOUNT**

&.PCOUNT is an integer ATV that is local to the macro it appears in. It contains the number of valid actual macro parameters that appeared on the macro call statement that called this macro. Formal parameters that have default values and are defaulted on the call are counted in &.PCOUNT. The label parameter is not counted.

Example:

A macro call statement that defaults two parameters

```
COPY 12,,ABC,&INT,,-1
```

the macro name statement:

```
COPY &P1,&P2,&P3,&P4,&P5'=D6',&P6
```

&.PCOUNT contains 5 since five parameters are used.



# Appendix K

## HP-System Macro Library

This Appendix describes the macros available in the HP System Macro Library. These are macros for commonly used operations such as calling and defining subroutines that use .ENTR, and executing conditional branches. This chapter describes what these macros are, what they do, and how best to use them.

### What the System Macros Do

The macros included in the library all have a number of things in common. They are usable in a wide range of programs, they hide some of the details of HP1000 assembly programming, and they generate assembly-language statements that get the job done in an efficient way. Most of the macros use conditional assembly to generate the best sequence of instructions possible given the information available at assembly time.

Most of the macros have parameters that specify memory locations (variables, literals, etc.) or registers to use in the macro expansion. Occasionally, though, a parameter is just a number, like an amount to shift or a bit position to test. It is important to understand what a particular parameter is used for, so that you do not use a 1 where you wanted an =D1 (the first a number, the second a memory location). Note that when a parameter specifies a register name, it must be either A or B, not 0, 1, X, Y, etc.

These macros generate assembly code that uses only those instructions found on all HP1000's: those in the memory-reference group, shift-rotate group, alter-skip group and instructions involving long shifts. The arithmetic macros do 16-bit integer arithmetic only, and they do not check for overflow. Some of these macros generate literals. The descriptions include sample generated code sequences; these sequences are specific to the parameters supplied, and to this version of the library. HP reserves the right to change the sequences generated as long as the external effects of the macros remain substantially the same.



## HP System Macro Library

Some of the macros generate labels or call other, inner macros to get things done. To avoid conflicts, do not use long labels that start with a lot of strange characters, like `#!#ELSE23`. Do not use macros whose names start with a period, like `.DOIFERROR`, and global assembly-time variables whose names start with `&.`, like `&.IFLEVEL`.

The descriptions of the macros include a summary of which registers may be changed. The E and O flags are always indeterminate after executing code generated by a macro; use these at your own risk. None of the macros touches X or Y.

The following is a list of all system macros described in this Appendix in order of appearance:

ENTRY	SETBIT
EXIT	CLEARBIT
CALL	TESTBIT
	FIELD
IF	
ELSE	ROTATE
ELSEIF	ASHIFT
ENDIF	LSHIFT
	RESOLVE
ADD	
SUBTRACT	TEXT
MAX	MESSAGE
MIN	
	TYPE
	STOP

## A Macro Example

The best way to explain how to use the macro library is to give an example. The following example shows using the library macros ENTRY, IF, ENDIF, and EXIT to write a Fortran-callable move words routine which uses MVW. The macros will be explained in detail later; only their context will be examined here.

```

1  macro,l
2      nam mvw,7,99 FORTRAN-callable .MVW &.date
3      maclib $MACLIB
4      GLOBALS
5  *
6  *  Routine to allow FORTRAN (or pascal, etc.) to use the
7  *  microcoded .MVW word-mover. Parameters are source,
8  *  destination, number of words.
9  *
10 mvw    ENTRY ^source,^destination,^count
11        IF @^count,>=d0    ; positive counts only
12        lda ^source
13        ldb ^destination
14        mvw @^count
15        ENDIF
16        EXIT
17        end

```

For this example, all references to system macros are in capital letters. Line 3 specifies using the system macro library \$MACLIB.

Line 4 calls the macro GLOBALS; this is a macro to define global assembly-time variables, which several of the macros use. It is best always to include this call to GLOBALS. If you ever get strange error messages from system macros, check that you have not left this out.

Note that you only need to have one MACLIB and one GLOBALS statement per file, even if you have multiple modules in one file. These statements work across module boundaries.

## HP System Macro Library

Line 10 is a call to ENTRY. It defines a routine MVW that should use the .ENTR entry sequence. It passes three parameters, called ^source, ^destination and ^count. These names start with a caret (^) as reminders that they are just pointers to the real arguments to the subroutine. ENTRY reserves locations for the parameters and return address, and calls .ENTR. It also generates the ENT MVW to make this a callable subroutine.

Line 11 is a run-time IF macro, not to be confused with the assembly-time AIF pseudo-op. It specifies that the lines following the IF macro and preceding the ENDIF macro should be executed only if the count passed was a positive number. The MVW routine requires a positive number for the count.

Lines 12, 13, and 14 are executed only if the count is greater than zero (@^count means to use ^count indirectly). Otherwise the code will jump to line 15. The commas around the condition enable the Macro Assembler to separate the parameters.

Line 15 is an ENDIF macro that generates a label for the jump generated by the IF macro.

Line 16 is an EXIT macro which generates the subroutine return instruction JMP @MVW.

While using the system macro library, you might want to take advantage of the listing options in Macro:

- |             |  |
|-------------|--|
| LIST short  | will list only the macro calls.  |
| LIST medium | will include the generated code and the calls to macros internal to the library. |
| LIST long   | will give the complete text of all macros.                                       |

## Description of System Macros

In the following descriptions, macro names are in upper case, parameters are in lower case, optional parameters are in brackets. Three dots indicates continued parameters.

### Subroutine Operations

#### Macro ENTRY:

```
subroutinename ENTRY [parameter1,...,parameter10]
```

Macro ENTRY generates the .ENTR entry sequence for subroutines. Subroutinename is declared an entry point. Destroys A and B. Refer to the EXIT macro regarding scope rules for ENTRY and EXIT. Up to ten parameters are allowed. After execution, pointers are set up to the parameters passed to the subroutine.

#### Example:

```
InitializeData ENTRY ^buffer,^inputlu
```

generates

```

^buffer      NOP
^inputlu     NOP
             ENT InitializeData
InitializeData NOP
             EXT .ENTR
             JSB .ENTR
             DEF ^buffer

```

## HP System Macro Library

### Macro EXIT:

EXIT

Macro EXIT generates the subroutine return jump appropriate for the most recent ENTRY. "Most recent" refers to the order in which the subroutines appear in the source, not the order in which they were called.

### Example:

Following the previous example for ENTRY, the statement:

EXIT

generates

JMP @InitializeData

## HP System Macro Library

### Macro CALL:

```
CALL subroutine[,parameter1,...,parameter10]
```

Macro CALL generates a call to an external subroutine, using the .ENTR calling sequence. Up to ten parameters are allowed. No registers are changed.

### Example:

```
CALL Namr,pbuf,@^buffer,len,start
```

generates

```
EXT Namr
JSB Namr
  DEF *+5
  DEF pbuf
  DEF @^buffer
  DEF len
  DEF start
```



Other forms of this macro are available which do not include the DEF to the return address ("direct" call), or which do not put an EXT on the subroutine name ("local" call):

Macro name	DEF return?	EXT on label?
-----	-----	-----
CALL	yes	yes
DCALL	no	yes
LCALL	yes	no
DLCALL	no	no

## Runtime Conditionals

### Macro IF

```
IF operand1,comparison,operand2
```

Macro IF generates a jump to a label if the comparison indicated is false. (The label is generated by ELSE, ELSEIF or ENDIF also.)

Operand 1 can be a memory location or a register. If it is a memory location, the A-Register will be used for the comparison.

Comparison specifies one of the six conditions below. Either the alphabetic (FORTRAN-like) or symbolic (Pascal-like) coding can be used.

Operand 2 should specify a memory location. The case of operand2 being =D0 is a special case, and causes efficient code to be generated. See the second example below.

Comparisons other than equality tests will probably generate a subtract to check the condition; this will change the register. The operands must be 16-bit integers; no overflow test is done.

IFs can be nested up to ten deep; this is discussed in more detail later.

<u>Comparison</u>	<u>Alpha</u>	<u>Sym</u>	<u>Register changed?</u>
equal	EQ	=	no
not equal	NE	<>	no
greater than	GT	>	yes
less than	LT	<	yes
greater than or equal	GE	>=	yes
less than or equal	LE	<=	yes

## HP System Macro Library

Example:

```
IF B,LT,BAZ
```

generates

```
CMB  
ADB BAZ  
SSB  
JMP name
```

Example:

```
IF Length,<>,=d0
```

generates

```
LDA Length  
SZA,RSS  
JMP anothername
```

IF is used with the macros, ELSE, ELSEIF, and ENDIF. IF generates a jump to a label which will be right after the next ELSE, ELSEIF or ENDIF. These macros communicate these labels through global assembly-time variables. You declared these labels at the beginning of the program with GLOBALS macro.



**Macro ELSE**

ELSE

Macro ELSE generates a jump to a label that will be defined by ENDIF, then defines the label that the previous IF macro referred to. Does not affect any registers.

Example:

ELSE

generates

```

        JMP anothername
name EQU *
```

ELSE is used in conjunction with IF and ENDIF. For example:

```

    IF A,LT,Tablesize
        ...
    ELSE
        ...
    ENDIF
```

The dots indicate omitted statements. This example causes the group of statements between the IF and ELSE macros to execute only if the content of the A-Register is less than the content of location Tablesize when the comparison code actually EXECUTES. If A is equal to or greater than Tablesize, the block of code between the ELSE and the ENDIF executes. The code before the IF and after the ENDIF executes in either case.

**Macro ELSEIF**

ELSEIF operand1,comparison,operand2

Macro ELSEIF generates an ELSE followed by an IF, with the IF using the supplied operands and comparison. They are exactly the same as they are on an IF macro. ELSEIF does not increase the nesting level. This means that many blocks of code can be separated by ELSEIFs without requiring a separate IF-ELSEIF pair for each. Otherwise it behaves as an ELSE followed by an IF. It must be followed at some point by an ELSE or an ENDIF.

Example:

```

IF A,EQ,'=S'RE'' ; be careful passing quoted string
                    (could use =A literal here, too)
    ...do RE command...

ELSEIF A,EQ,'=S'SE''
    ...do SE command...

ELSE ;             note that elses are optional!
    ...here if we didn't want to do the others...

ENDIF

```

generates

```

CPA =S'RE'          ; IF
JMP *+2
JMP test2
    ...do RE command...

JMP endoftests     ; ELSEIF
test2 CPA =S'SE'
JMP *+2
JMP test3
    ...do SE command...

JMP endoftests     ; ELSE
test3 EQU *
    ...here if we didn't want to do the others...

endoftests EQU *   ; ENDIF

```

**Macro ENDIF**

ENDIF

Macro ENDIF generates the label that IF, ELSE or ELSEIF wants to jump to. Generates no code. It reduces the nesting level by one (see example below).

Example:

```
IF Error,>=,=d0
    IF Length,=,=d-1    ; note nested if
        ...have end of file...
    ELSE
        ...do something
    ENDIF                ; return to outer nesting level
ELSE
    ...have negative error code...
ENDIF
```

Of course "nesting" refers to the code executed as a result of the various tests, not to the indentation of the text! In the above example, the end-of-file test is made only the error code was not negative. Remember the nesting level is limited to ten levels; ELSEIF is good to use to keep nesting levels from getting out of hand.

## Arithmetic Operations

### Macro ADD

```
ADD source,amount[,destination]
```

Macro ADD generates code to add amount, which is a memory location, to source, which is a memory location or register. If a memory location is specified, the A-Register will be used for the add if a register is needed. The result is left in the register used, unless a destination to store to is provided. As always, it works only with 16-bit integers, and does not check for overflow.

#### Example:

```
ADD Value,=d4,NewValue
```

generates

```
LDA Value
ADA =d4
STA NewValue
```

#### Example:

```
ADD Loc,=d1,Loc
```

generates

```
ISZ Loc      ; no register needed!
NOP
```

**Macro SUBTRACT**

```
SUBTRACT source,amount[,destination]
```

Macro SUBTRACT generates code to subtract amount, which is a memory location, from source, which is memory location or a register. If a memory location is specified, the A-Register will be used for the operation. The result is left in the register used, unless a destination is specified. It uses 16-bit integers, and does not check for overflow.

Example:

```
SUBTRACT @Pointer,=d16
```

generates

```
LDA =d-16  
ADA @Pointer
```

Example:

```
SUBTRACT Value,Base,Offset
```

generates

```
LDA Base  
CMA,INA  
ADA Value  
STA Offset
```

**Macro MAX**

MAX operand1,operand2

Macro MAX generates code to find the maximum of the two memory locations operand1 and operand2. The A-Register is used in the computation, so it will not do the right thing if the A-Register is used as either operand. The result is left in the A-Register. It uses 16-bit integers, and does not check for overflow.

Example:

MAX SupplyVoltage,=d500

generates

```
LDA SupplyVoltage
CMA,INA
ADA =d500
SSA
CLA
ADA SupplyVoltage
```

**Macro MIN**

MIN operand1,operand2

Macro MIN generates code to find the minimum of the two memory locations operand1 and operand2. The A-Register is used in the computation, so it will not do the right thing if the A-Register is used as either operand. The result is left in the A-Register. It uses 16 bit integers, and does not check for overflow.

Example:

MIN SupplyVoltage,=d500

generates

```
LDA SupplyVoltage
CMA,INA
ADA =d500
SSA,RSS
CLA
ADA SupplyVoltage
```

## Bit Operations

### Macro SETBIT

SETBIT bitnumber

Macro SETBIT generates code to set bit bitnumber in the A-Register. Bitnumber should be just an ordinary number, not a memory location. Bit 0 is the least significant bit, bit 15 is the most significant (the sign bit). The IOR instruction is used to set the bit.

Example:

SETBIT 6

generates

IOR =D64



**Macro CLEARBIT**

CLEARBIT bitnumber[,register]

Macro CLEARBIT Generates code to clear the specified bit in the A-Register, or if the bit number is 0 or 15, the B-Register can be specified as the register to use. Bitnumber must be just a number, not a memory location. Bit 0 is the least significant bit, bit 15 is the most significant (the sign bit). It uses the AND instruction or an SRG instruction to clear the bit.

Example:

CLEARBIT 9

generates

AND =D-513

Example:

CLEARBIT 15,B

generates

ELB,CLE,ERB

## HP System Macro Library

### Macro TESTBIT

TESTBIT location,bitnumber,value,instruction

Macro TESTBIT generates code to execute the specified instruction only if the bitnumber in the location specified is currently equal to value.

Location is either a memory location or a register. If it is a memory location, the A-Register will be used for the test; if it is the B-Register, and the specified bit cannot be tested easily, the A-Register will be used. (In short, this macro destroys the A-Register.)

Bitnumber is the bit to test, and is a number, 0 through 15. Bit 0 is the least significant, bit 15 is the most significant (the sign bit).

Value is the bit value to test for; it is either 0 (a number), or is any other value, meaning non-zero.

Instruction is a statement to execute if the specified bit has the specified value. It will almost always be a jump instruction, and so will have to be in quotes: 'JMP target'. The instruction cannot have a label.

Example:

```
TESTBIT A,4,0,'JMP AWAY'
```

generates

```
AND =D16
SZA,RSS
JMP AWAY
```

Example:

```
TESTBIT B,0,1,'ALF,ALF' ; ALF is in quotes to defeat comma
```

generates

```
SLB
ALF,ALF
```

**Macro FIELD**

FIELD location,startbit,fieldwidth

Macro FIELD generates code to isolate fieldwidth bits starting at bit startbit from location, leaving them right-justified in the A-Register.

Location can be a memory location or a register; the A-Register will be used for the operation in any case.

Startbit is a number, 0 through 15; bit 0 is the least significant bit; bit 15 is the most significant bit (sign bit). This is the right-most bit of the bit field desired.

Fieldwidth is a number specifying how many bits wide the field should be. Startbit + fieldwidth must not be greater than 16.

Example:

```
FIELD A,8,8 ; extract upper byte
```

generates

```
ALF,ALF ; rotate 8  
AND =D255 ; and mask
```

Example:

```
FIELD EQT5,14,2
```

generates

```
LDA EQT5  
RAL,RAL  
AND =D3 ; keep just availability code.
```

## Shifts

### Macro ROTATE



ROTATE location,distance

Macro ROTATE generates code to do rotation of location by distance bits.

Location can be a memory location or a register; if a memory location is specified, the A-Register is used to do the rotate. The result is left in the register used.

Distance is the number of bits to rotate. It is a number, from -16 to +16. Positive numbers mean rotate left, negative numbers mean rotate right. 16 bits are rotated.

The other register is never touched, regardless of distance; all rotates can be done in one or two instructions (not counting the initial load).

Example:

```
ROTATE B,4
```

generates

```
BLF
```

Example:

```
ROTATE Flag,-3
```

generates

```
LDA Flag  
RAR,RAR  
RAR
```

**Macro ASHIFT**

ASHIFT location,distance

Macro ASHIFT generates code to do an arithmetic shift of location by distance bits. (Arithmetic shifts propagate the sign bit.)

Location can be a memory location or a register. If a memory location is specified, the B-Register is used to do the shift. The result is left in the register used.

Distance is the number of bits to shift. It is a number, from -16 to +16. Positive numbers mean shift left, negative numbers mean shift right. 16 bits are shifted.

If the distance to shift is greater than 2 or less than -2, the content of the other register will be destroyed.

Example:

ASHIFT B,4

generates

CLA  
ASL 4 ; result in B

Example:

ASHIFT A,-2

generates

ARS,ARS

**Macro LSHIFT**

LSHIFT location,distance

Macro LSHIFT Generates code to do a logical shift of location by distance bits. Location can be a memory location or a register; if a memory location is specified, the B-Register is used to do the shift. The result is left in the register used.

Distance is the number of bits to shift. It is a number, from -16 to +16. Positive numbers mean shift left, negative numbers mean shift right. 16 bits are shifted.

If the distance to shift is greater than 2 or less than -3, the content of the other register will be destroyed.

Example:

LSHIFT B,4

generates

CLA  
LSL 4 ; result in B

Example:

LSHIFT A,-2

generates

CLE,ERA  
ARS

**Macro RESOLVE**

RESOLVE register

Macro RESOLVE generates code to do resolution of indirects on an address in the A- or B-Register. It assumes the register contains a pointer to a DEF. It is designed for use in parameter passing on direct calls that do not want to use .ENTR.

Example:

```
RESOLVE A
```

generates

```
LDA @A  
RAL,CLE,SLA,ERA  
JMP *-2
```

## Text Definition

### Macro TEXT

TEXT string

Macro TEXT generates an ASC pseudo op for the given quoted string. Designed to be used in the data definition part of a program.

Example:

```
TEXT 'Hello, how are you?'
```

generates

```
ASC 10,Hello, how are you?
```

### Macro MESSAGE

MESSAGE pointer,text

Macro MESSAGE generates a block of memory containing the length of the given string in words, followed by the string. Pointer is a pointer to the length word. This is designed for calls to output routines that need a string length (WRITE, EXEC).

Example:

```
MESSAGE ^errmsg,'No such file'
```

generates

```
^errmsg DEF *+1  
DEC 6  
ASC 6,No such file
```



## Communication with RTE

### Macro TYPE

TYPE message

Macro TYPE generates an EXEC 2 call to send a message to the terminal. It uses LU 1 as the terminal to talk to, so it is useful mostly in systems with session monitor. Message is a quoted string.

Example:

```
TYPE 'All tests completed. Everything OK!'
```

generates

```
EXT EXEC
JSB EXEC
  DEF +*5
  DEF =D2
  DEF =D1
  DEF =S'All tests completed. Everything OK!'
  DEF =L-36
```

### Macro STOP

STOP

Macro STOP generates an EXEC 6 call to stop the current program. This is a normal stop, and releases resources, etc.

Example:

```
STOP
```

generates

```
JSB EXEC
  DEF +*2
  DEF =D6
```

# Appendix L

## Backward Compatible Constructs

This section contains constructs of Macro/1000 that exist to support backward compatibility with earlier HP assemblers. An alternate way to use all of these constructs is described in the main body of this manual.

### Assembler Control Statement

Macro allows the old control statement ASMB instead of MACRO to be used. All assembly option parameters described in Appendix E may be on this statement in addition to the ones described below:

**N,Z Selective Assembly Options:** parameters govern the state of the IFN/IFZ options (see below).

**P Override Option:** used as an override option when MACRO is invoked. It has no effect when specified in the control statement of an assembly language program. This option was used to make the previous assembler backward compatible to the one before it.

**B,F,X, Ignored if Specified:** included here for reasons of backward compatibility.

## Indirection Indicator

To maintain backward compatibility, the letter I can be appended to a label to indicate indirection. It is separated from the label by a comma. If an indirect label is used as a parameter to a macro and the indirection indicator is ',I', then Macro will interpret the 'I' as the next parameter in the Macro's parameter list. User's who use ',I' are cautioned to watch for opcodes that are redefined as macros.

The new way to indicate indirection is to append @ (at sign) to the label.

## Clear Flag Indicator

The old way of indicating that the flag should be cleared on an I/O instruction is maintained for old code. You can append a ',C' to the operand in the I/O instructions. The 'C' is then not needed on the opcode. The new way to set the clear flag is to append the 'C' to the opcode itself.

## Old Literal Constructs

The =A literal places ASCII characters into the literals table. The new way to do this is by use of the =S literal.

## Old Pseudo Ops

The pseudo ops discussed in this section are:

ORB	XIF	UNL
ORR	REP	LST
IFN	COM	MIC
IFZ	EMA	RAM

### ORB

Syntax:

```
ORB [;comments]
```

ORB defines the portion of a relocatable program that must be assigned to the base page by Macro/1000. The label field (if given) is ignored and the statement requires no operand. All statements that follow the ORB statement are assigned contiguous locations in the base page. Assignment to the base page terminates when Macro/1000 detects an ORG, ORR, RELOC, or END statement.

When more than one ORB is used in a program, each ORB causes Macro to resume assigning base page locations at the address following the last assigned base page location. For example:

```

        NAM PROG      ; Assign zero as relative starting location for
        :             ; program PROG.
        :             ;
        ORB           ; Assign all following statements to base page.
IAREA BSS 10        ; Reserve 100 locations on base page.
        :
        ORR          ; Continue main program.
        :
        ORB          ; Resume assignment at next available location ??
        :             ; base page.
        :
        ORR          ; Continue main program.
    
```

An ORB statement in an absolute program has no significance and is flagged as an error.

The new way to assemble onto base page is to replace the ORB statement with a RELOC BASE statement.

## Backward Compatible Constructs

### ORR

Syntax:

```
ORR [;comments]
```

ORR resets the program location counter to the value existing when an ORG, or ORB instruction was encountered. For example:

```
      NAM RSET          ; Set PLC to value of zero, assign RSET as
FIRST ADA              ; name of program.
      :
      ADA CTRL         ; Assume PLC at FIRST+2280.
      ORG FIRST+2926  ; Save PLC value of FIRST+2280 and set PLC to
      :               ; FIRST+2926.
      :
      JMP EVEN+1      ; Assume PLC at FIRST+3004.
      ORR             ; Reset PLC to FIRST+2280.
```

More than one ORG statement can occur before an ORR. If so, when the ORR is encountered the program location counter is reset to the value it contained when the first ORG of the set occurred. For example:

```
      NAM RSET          ; Set PLC to zero.
FIRST ADA              :
      LDA WYZ          ; Assume PLC at FIRST+2250.
      ORG FIRST+2250  ; Set PLC to FIRST+2500.
      :
      LDB ERA          ; Assume PLC at FIRST+2750.
      ORG FIRST+2900  ; Set PLC to FIRST+2900.
      :
      CLE              ; Assume PLC at FIRST+2920.
      ORR             ; Reset PLC to FIRST+2250.
```

If a second ORR appears before an intervening ORG or ORB the second ORR is ignored.

The new way to perform an ORR is to use the RELOC command with the appropriate keyword (PROG, COMMON, BASE, EMA, SAVE, and CODE).

## IFN, IFX, and XIF

Syntax:

```
IFN [;comments]
IFX [;comments]
XIF [;comments]
```

The IFN and IFZ pseudo instructions cause the inclusion of instructions in a program provided that either an 'N' or 'Z', respectively, is specified as a parameter for the control statement (discussed earlier in this Appendix). The IFN or IFZ instruction precedes the set of statements that are to be included. The pseudo instruction XIF serves as a terminator to both the set of statements and the assembly.

```
IFN
:
XIF
```

All source language statements appearing between the IFN and the XIF pseudo instructions are included in the program if the character 'N' is specified on the control statement.

All source language statements appearing between the IFZ and XIF pseudo instructions are included in the program if the character 'Z' is specified on the control statement.

```
IFZ
:
XIF
```

When the particular letter is not included on the control statement, the related set of statements appears on the assembler output listing but is not assembled.

Any number of IFN-XIF and IFZ-XIF sets can appear in a program; however, they cannot overlap. An IFZ or IFN intervening between an IFZ or IFN and the XIF terminator results in a diagnostic being issued during compilation; the second pseudo instruction is ignored.

Both IFN-XIF and IFZ-XIF pseudo instructions can be used in the program; however, only one type will be selected in a single assembly. Therefore, if both characters 'N' and 'Z' appear in the control statement, the character listed last will determine the set of coding that is to be assembled.

## Backward Compatible Constructs

A more general way of doing conditional assembly is through use of the AIF statement. The selection may be done in the run string by initializing the system run string assembly-time variables, and using those variables in AIF statements. For example:

```
:RU,ASMB,&S,-,-,,N  
  
  NAM TEST  
  :  
  IFN  
    sequence of statements  
  XIF  
  :  
  END
```

This can be accomplished in following way:

```
:RU,MACRO,&S,-,-,,, &.RS1=N ('N' is the initial value for r &.RS1)  
  
  NAM TEST  
  :  
  AIF &.RS1 = 'N'  
    sequence of statements  
  AENDIF  
  :  
  END
```

## Backward Compatible Constructs

### REP

Syntax:

```
REP n [;comments]
```

The REP pseudo instruction causes the repetition of the statement immediately following it a specified number of times.

The statement following the REP in the source program is repeated n times. The n may be any absolute expression. Comment is ignored and the instruction following the comment is repeated.

A label specified in the REP pseudo instruction is assigned to the first repetition of the statement. A label should not be part of the instruction to be repeated, it would result in a double defined symbol error.

Example:

```
      CLA
TRIPL REP 3
      ADA DATA
```

This would generate the following:

```
      CLA          Clear the A-Register; The content of DATA
TRIPL ADA DATA  is tripled and stored in the A-Register.
      ADA DATA
      ADA DATA
```

Example:

```
FILL REP 100B
      NOP
```

This loads 100B memory locations with the NOP instruction. The first location is labeled FILL.

Do not mix the REP statement with new Macro constructs. Macro calls must not be the statements to be repeated with the REP.

The new way of doing this is through use of the REPEAT statement.



**COM**

Syntax:

```
COM name1[(size1)][,name2[(size2)],...,namen[(sizen)]][;comments]
```

COM reserves a block of storage locations that can be used in common by several subprograms. Each name identifies a segment of the block for the subprogram in which the COM statement appears. The sizes are the number of words allotted to the related segments. The size is specified as an octal or decimal integer. If the size is omitted, it is assumed to be 1.

Any number of COM statements can appear in a subprogram. Storage locations are assigned contiguously. The length of the block is equal to the sum of the lengths of all segments named in all COM statements in the subprogram.

To refer to the common block, other subprograms must also include a COM statement. The segment names and sizes may be the same or they may differ. Regardless of the names and sizes specified in the separate subprograms, there is only one common block for the combined set. It has the same relative origin; the content of the n-th word of common storage is the same for all subprograms. For example:

```
PROG1 COM ADDR1(5),ADDR2(5),ADDR3(5)
      :
      LDA ADDR2+1 ; Pick up second word of array ADDR2.
      :
      END
      :
PROG2 COM AAA(2),AAB(2),AAC,AAD(10)
      :
      LDA AAD+1 ; Pick up second word of array ADD.
```

## Backward Compatible Constructs

### Organization of Common Block

PROG1 name	PROG2 name	Common block
ADDR1	AAA	location 1
		location 2
	AAB	location 3
		location 4
ADDR2	AAC	location 5
	AAD	location 6
		location 7
		location 8
ADDR3		location 9
		location 10
		location 11
		location 12
		location 13
		location 14
		location 15

The segment names that appear in the COM statements can be used in the operand fields of DEF, ABS, EQU, ENT or any memory reference statement; they cannot be used as labels elsewhere in the program.

The loader establishes the origin of the common block; the origin cannot be set by the ORG pseudo instruction. All references to the common area are relocatable.

Two or more subprograms can declare common blocks that differ in size. The subprogram that defines the largest block must be the first submitted for loading.

The new way to perform the COM is through the use of the RELOC COMMON statement, followed by BSS's.

## EMA

Syntax:

```
label EMA m1,m2 [;comments]
```

The EMA pseudo instruction defines an extended memory area (EMA) where m1 is the EMA size in pages and m2 is the mapping segment (MSEG) size in pages. m1 and m2 must be expressions that evaluate to non-relocatable integers. m1 must be in the range 0 to 1023 inclusive and m2 must be in the range 0 to 31 inclusive. If m1 evaluates to 0 the maximum possible size for MSEG will be assigned at load time.

The EMA pseudo instruction can be used only in a relocatable program. More than one EMA pseudo instruction is not allowed.

THE EMA COMMAND CANNOT BE USED IN THE SAME PROGRAM WITH A RELOC EMA COMMAND OR AN ALLOC EMA COMMAND. An EMA pseudo instruction must have been assigned to the storage area. This label represents the logical address of the first word in the MSEG and is determined at load time. EMA labels can appear in memory reference statements, and in EQU or DEF pseudo instructions.

References to EMA labels are processed as indirect addresses through a base page link at load time. EMA labels can be referenced in other subprograms or segments by declaring them as externals in the other subprograms or segments. Do not declare them as entry points in the program in which they appear.

The following restrictions apply to the use of EMA labels:

1. EMA labels cannot be used with an offset.
2. EMA labels cannot be used with indirect.
3. EMA labels cannot appear in an ENT or COM statement in the same subprogram.

The following example illustrates the use of a forty-page EMA that has a five-page MSEG. In the example, the main program calls MMAP to map the third MSEG into its logical address space. Then it stores the value at the start of the third MSEG into the 1028th word of the third MSEG. Then it calls a subroutine to process that element. The subroutine loads the value into the B-Register to process it, and returns to the calling program. Refer to Figure L-1 for a pictorial explanation of the elements that are being addressed.

## Backward Compatible Constructs

```

        NAM EMAPR,3
        EXT EMASB
        EXT MMAP
EMALB EMA 40,5 ; 40 pages of EMA, 5 pages per MSEG.
ADEMA DEF EMALB
*
* Call MMAP to map third MSEG into programs logical address space.
*
EMAPR JSB MMAP
      DEF RTN
      DEF IPGS ; Offset in pages of MSEG being mapped.
      DEF NPGS ; Number of pages in MSEG.
*
* Store first word of third MSEG into 1028th word of third MSEG.
*
RTN   LDA EMALB ; First word of MSEG referenced directly.
      LDB ADEMA ; Use B-Reg. to reference 1028th word of MSEG.
      RBL,CLE,SLB,ERB ; Resolve one indirect.
      LDB @B
      ADB =D1027
      STA @B ; Store A-Reg. into 1028th word of current MSEG.
*
* JSB passing offset address to subroutine.
*
      JSB EMASB
      DEF =D1027 ; Pass offset address as parameter.
      :
IPGS DEC 10
NPGS DEC 5

      NAM EMASB,7 ; External subroutine and segments.
      ENT EMASB ; Declare EMA as an external.
      EXT EMALB
ADEMA DEF EMALB
*
* Subroutine entry point is here, A-Reg. is used to compute address.
*
EMASB NOP
      LDA EMASB ; Get address of offset.
      LDA @A ; Get offset value.
      LDB ADEMA
      RBL,CLE,SLB,ERB
      LDB @B
      ADA @B ; Add is address of current MSEG start.
      LDB @B ; Load 1028th word of current MSEG into B-Reg.
      :
      ISZ EMASB
      JMP EMASB ; Return to caller.
      END

```

# Backward Compatible Constructs

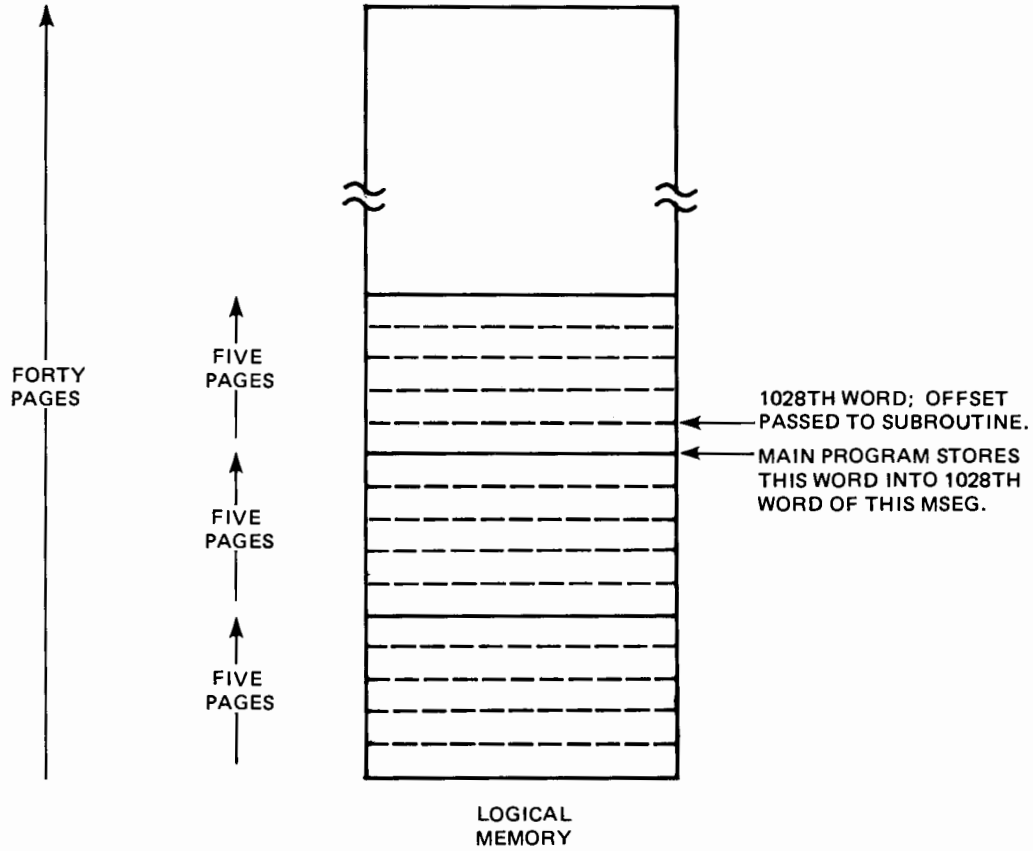


Figure L-1. Pictorial Explanation of Elements Being Addressed

The new way to declare EMA space is to use the ALLOC EMA statement.

## UNL

Syntax:

```
UNL [;comments]
```

List output is suppressed from the assembly listing, beginning with the UNL pseudo instruction and continuing for all instructions and comments until either an LST or an END pseudo instruction is encountered. Diagnostic messages for errors encountered by Macro/1000 will be printed, however. The source statement sequence numbers (printed in columns 1-5 of the source program listing) are incremented for the instructions skipped.

The new way to do this is by using the LIST OFF command.

## LST

Syntax:

```
LST [;comment]
```

The LST pseudo instruction causes the source program listing, terminated by a UNL, or a LIST OFF command, to be resumed.

A UNL following a UNL, an LST following an LST, and an LST not preceded by a UNL are not considered errors by Macro/1000. Macro counts the number of LST and UNL statements it encounters. Only when the count is equal will it change the current listing state. In other words, if the program has three LST statements with no UNL statements in between them, only until the fourth UNL statement is given will the listing be turned off.

The new way to do this is by using the LIST command.



**MIC**

Syntax:

```
MIC opcode, fcode, pnum [;comments]
```

The MIC pseudo instruction allows you to define your own instructions. The opcode is a 1- to 16-character mnemonic, fcode is an instruction code, and pnum declares how many (0-7) parameter addresses are to be associated with the newly-defined instruction.

Both fcode and pnum may be expressions that generate an absolute result. A user-defined instruction must not appear in the source program prior to the MIC pseudo instruction that defines it. When the user-defined mnemonic is used later in the source program, the specified number of parameter addresses (pnum) is supplied in the operand field of the user-defined instruction, separated by spaces.

The parameter addresses can be any addressable values, relocatable and/or indirect. The parameters cannot be literals, assembly-time variables or macro parameters.

All three operands (opcode, fcode, and pnum) must be supplied in the MIC pseudo instruction in order for the specified instruction to be defined. If pnum is zero, it must be expressly declared as such (not omitted).

Example - "Jump to Microprogram"

The MIC pseudo instruction is primarily intended to facilitate the passing of control from an assembly language program to a user's microprogram residing in Read Only Memory (ROM) or Writable Control Store (WCS). Ordinarily, to do this you must include an OCT 101xxx or OCT 105xxx statement (where xxx is 140 thru 737) at the point in the source program where the jump is to occur. If parameters are to be passed, they are usually defined as constants (via OCT, DEC, DEX, DEY, or DEF statements) immediately following the OCT 105xxx statement.

With the MIC pseudo instruction, you can define a source-language instruction that passes a series of additional parameters to a microprogram beyond those pointed to by the user-defined instruction. The parameters must be defined as constants (via OCT, DEC, DEX, DEY, or DEF statements) immediately following each use of the user-defined instruction.

## Backward Compatible Constructs

### Example - Microprogram Example

For example, assume that the first two parameters to be passed from the assembly-language program to your microprogram reside in memory locations PARM1 and PARM2, and that the third parameter resides in the memory location pointed to by ADR. Also assume that the octal code for transferring control to the particular microprogram is 105240B.

The following statement defines a source-language instruction that passes control and three parameter addresses to the microprogram.

```
MIC ABC,105240B,3
```

Whenever you want to pass control from the assembly-language program to the microprogram, you can use the following user-defined instruction in the source program:

```
ABC PARM1 PARM2 ADR
```

The new way to do this is with the RPL instruction.

## RAM

Syntax:

```
RAM m[;comments]
```

An alternate but somewhat restricted way to access microprogrammed functions from the Assembler language is by employing the RAM (Random Access Memory) pseudo-instruction. The RAM pseudo-instruction will generate an executable machine instruction which when executed will cause a jump to microcode. The high order bits of the instruction will be 105 octal and low order bits will be the octal value of m. m must evaluate to an absolute expression in the range 0 to 377 octal.

Example, the following lines of assembly code:

```
RAM B16  
B16 EQU 16B
```

will generate this octal object code:

```
105016
```





# Index

&.ERROR, J-1, J-2  
 &.PCOUNT, J-1, J-4  
 &.Q, 5-13, J-1  
 &.REP, J-1, J-3  
 &.RS1, E-8, J-1, J-3  
 &.RS2, E-8, J-1, J-3

:AND: (logical AND), 4-66  
 :ASH:, 4-63, 4-64  
 :L: (length attribute), 4-59  
 :LSH:, 4-63, 4-64  
 :MOD:, 4-63, 4-64  
 :NOT:, 4-59  
 :OP:, 5-7  
 :OR: (logical OR), 4-66  
 :ROT:, 4-63  
 :S: (substring attribute), 4-59  
 :T: (type attribute), 4-59  
 :UC: (upper case attribute), 4-59

## A

ABS, 4-45, 4-48, B-16, C-1  
 absolute assembly, E-2  
 absolute code, 1-1, 1-12  
 absolute expressions, 2-11  
 absolute program, 2-11, 4-9, 4-13  
 absolute space, 4-2, 4-4  
 absolute value, 4-48  
 actual macro parameters, 5-3, 5-13  
 ADA, 3-2, B-2, C-1  
 ADB, 3-2, B-2, C-1  
 ADD (system macro), K-2, K-13  
 addition, 2-12  
 address and symbol definition, B-16  
 address definition instructions, 4-1, 4-45, B-1, B-16  
 ADX, 3-12, B-7, C-1  
 ADY, 3-12, B-7, C-1  
 AELSE, 4-68, B-17, C-1  
 AELSEIF, 4-68, B-17, C-1  
 AENDIF, 4-68, B-17, C-1  
 AENDWHILE, 4-68, B-17, C-1  
 AIF, 4-68, B-17, C-1  
 ALF, 3-7, B-4, C-1

alias, 4-19, 4-20  
ALLOC, 4-19, 4-21, 4-22, 4-47, B-15, C-1  
ALR, 3-7, B-4, C-1  
ALS, 3-7, B-4, C-1  
alter-skip group, 3-10, B-5  
AND, 3-2, B-2, C-1  
arithmetic negation, 4-60  
arithmetic operators, 2-11, 4-59, 4-63, 4-68  
ARS, 3-7, B-4, C-1  
ASC, 4-37, B-16, C-1  
ASCII characters, 4-37  
ASHIFT (system macro), K-2, K-22  
ASL, 3-14, B-10, C-1  
ASMB assembly language, 1-2  
ASR, 3-14, B-10, C-1  
assembler control instructions, 4-1, 4-2, 4-4, B-1, B-14  
assembler instructions, 1-11  
assembler instructions (see pseudo ops), 4-1  
assembly listing control instructions, 4-25  
assembly-time arrays, 4-57  
Assembly-Time Variable Declaration, B-1, B-17  
assembly-time variable, local, 4-56  
assembly-time variables (ATVs), 1-13, 2-8, 4-54, 4-55  
assembly-time variables, global, 4-56  
asterisk, 2-7  
AWHILE, 4-68, B-17, C-1

## B

backslash, 2-15  
backward compatibility, 1-2, L-1  
base page fence, 3-23  
base page relocatable space, 1-12, 4-2, 4-3, 4-11  
binary codes, D-1  
bit-processing, 3-4  
BLF, 3-7, B-4, C-1  
BLR, 3-7, B-4, C-1  
BLS, 3-7, B-4, C-1  
body of the macro definition, 5-2  
BRS, 3-7, B-4, C-1  
BSS, 4-35, B-16, C-1  
BYT, 4-37, 4-39, B-16, C-1  
byte processing, 2-12, 4-51  
byte-processing, 3-4

## C

CALL (system macro), K-2, K-7  
CALL subroutine operations, K-5  
calling macros, 5-1, 5-3

CAX, 3-11, B-7, C-2  
 CAY, 3-11, B-7, C-2  
 CBS, 3-6, B-3, C-2  
 CBT, 3-4, B-3, C-2  
 CBX, 3-11, B-7, C-2  
 CBY, 3-11, B-7, C-2  
 CCA, 3-10, B-5, C-2  
 CCB, 3-10, B-5, C-2  
 CCE, 3-10, B-5, C-2  
 CGLOBAL, 4-55, 4-58, B-17, C-2  
 CLA, 3-10, C-2  
 CLB, 3-10, B-5, C-2  
 CLC, 3-16, B-8, C-2  
 CLCC, B-8, C-2  
 CLE, 3-7, 3-10, B-4, B-5, C-2  
 clear flag indicator, L-2  
 CLEARBIT (system macro), K-2, K-18  
 CLF, 3-16, B-8, C-2  
 CLO, 3-17, B-9, C-2  
 CLOCAL, 4-55, 4-58, B-17, C-2  
 CMA, 3-10, B-5, C-2  
 CMB, 3-10, B-5, C-2  
 CME, 3-10, B-5, C-2  
 CMW, 3-5, B-3, C-2  
 COL, 4-25, B-15, C-2  
 COM, C-2, L-8  
 comment field, 1-7, 2-1, 2-2, 2-13  
 comments in macro definitions, 5-10  
 common relocatable space, 1-12, 4-2, 4-3, 4-11, 4-21  
 comparison operators, 4-59, 4-65, 4-68  
 concatenation, 4-59, 4-66  
 conditional assembly, 1-13, 4-1, 4-68, B-1, B-17  
 constant definition instructions, 4-1, 4-37, B-1, B-16  
 control statement, 1-2, 1-3, 1-4, 5-19, E-2, L-1  
 CPA, 3-2, B-2, C-2  
 CPB, 3-2, B-2, C-2  
 creating macro libraries, 5-1, 5-19  
 cross reference table, 1-10  
 cross-reference table, E-2, F-1  
 CSET, 4-58, B-17, C-2  
 CXA, 3-11, B-7, C-2  
 CXB, 3-11, B-7, C-2  
 CYA, 3-11, B-7, C-2  
 CYB, 3-11, B-7, C-2

## D

DBL, 2-12, 3-5, 4-45, 4-51, B-16, C-2  
 DBR, 2-12, 3-5, 4-45, 4-51, B-16, C-2  
 DDEF, 2-12, 4-45, 4-47, B-16, C-2  
 DEC, 4-37, 4-40, B-16, C-2

decimal constants, 4-40  
decimal integer, 4-40  
declaring assembly-time variables, 4-1, 4-54  
DEF, 2-12, 4-45, B-16, C-2  
default macro parameters, 5-15  
descriptions of system macros, K-5  
destination file, E-6  
DEX, 4-37, 4-41, B-16, C-2  
DEY, 4-37, 4-41, B-16, C-2  
DIV, 3-13, B-10, C-2  
DJP, 3-22, B-12, C-2  
DJS, 3-22, B-12, C-2  
DLD, 3-13, B-10, C-2  
DST, 3-14, B-10, C-2  
DSX, 3-11, B-7, C-2  
DSY, 3-11, B-7, C-2  
dynamic mapping system, 3-1, 3-23  
dynamic mapping system instructions, 3-19, B-1, B-12

## E

ELA, 3-7, B-4, C-3  
ELB, 3-7, B-4, C-3  
ELSE (system macro), K-2, K-10  
ELSEIF (system macro), K-2, K-11  
EMA, 4-11, C-3, L-10  
EMA programming, 4-36  
EMA relocatable space, 1-12, 2-12, 4-2, 4-3, 4-21, 4-35, 4-47  
END, 1-4, 4-4, 4-13, 4-33, B-14, C-3  
ENDIF (system macro), K-2, K-12  
ENDMAC, 5-2, 5-10, B-18, C-3  
ENDREP, 4-68, 4-73, B-17, C-3  
ENT, 4-19, B-15, C-3  
ENTRY (system macro), K-2, K-5  
ENTRY subroutine operations, K-5  
EQU, 4-45, 4-49, B-16, C-3  
ERA, 3-7, B-4, C-3  
ERB, 3-7, B-4, C-3  
error messages, A-1  
error reporting, 1-9, B-18  
evaluation of expressions, 4-69  
example of a macro, 5-2  
EXIT (system macro), K-2, K-6  
EXIT subroutine operations, K-5  
expressions, 2-4, 2-11, 2-12  
expressions using assembly-time variables, 4-59  
expressions, legal use of, 2-12  
EXT, 4-19, B-15, C-3  
Extended Arithmetic Group, 3-13  
extended arithmetic instruction, 3-1  
extended arithmetic unit, B-1, B-10

extended instruction group, B-1, B-7  
extended precision constants, 4-41  
extended relocatable records, 1-2, 1-3

## F

FAD, 3-18, B-11, C-3  
FDV, 3-18, C-3  
FIELD (system macro), K-2, K-20  
FIX, 3-19, B-11, C-3  
floating point instructions, 3-1, 3-18  
floating-point instructions, B-1, B-11  
floating-point numbers, 4-40  
FLT, 3-19, B-11, C-3  
FMP, 3-18, B-11, C-3  
formal macro parameters, 5-3, 5-6, 5-11  
FSB, 3-18, B-11, C-3  
FSV, B-11

## G

GEN, 4-17, 4-18, B-14, C-3  
generating microcode instructions, E-2  
generating old records, E-2  
generator control instructions, 4-1, 4-17  
global assembly-time variable, 4-56

## H

halt instruction, 3-1, 3-15, B-1, B-8  
heading, 4-26  
HED, 4-25, 4-26, B-15, C-3  
HLT, 3-17, B-9, C-3  
HLTC, B-9, C-3

## I

IF (system macro), K-2, K-8  
IFN, C-3, L-5  
IFX, L-5  
IFZ, C-3  
IGLOBAL, 4-55, 4-58, B-17, C-3  
ILOCAL, 4-55, 4-58, B-17, C-3  
implementation notes, I-1  
INA, 3-10, B-5, C-3  
INB, 3-10, B-5, C-3  
INCLUDE statement, 1-14, 4-15, 5-1, B-14, C-3  
index register instructions, 3-1, 3-11, B-7

indirect addressing indicator, 2-14, 3-1  
indirection addressing indicator, L-2  
input/output instructions, 3-1, 3-15, B-1, B-8  
input/output, overflow and halt, B-8  
instruction replacements, 3-24  
integer comparison, 4-65  
integer numbers, 4-40  
IOR, 3-2, B-2, C-3  
ISET, 4-58, B-17, C-3  
ISX, 3-11, B-7, C-3  
ISY, 3-11, B-7, C-3  
ISZ, 3-3, B-2, C-3

## J

JLY, 3-12, B-7, C-3  
JMP, 3-3, B-2, C-3  
JPY, 3-12, B-7, C-3  
JRS, 3-19, B-12, C-3  
JSB, 3-3, B-2, C-3

## L

label field, 2-1, 2-3, 5-5, 5-11, 5-13  
LAE, 3-7, B-4, C-4  
LAX, 3-12, B-7, C-4  
LAY, 3-12, B-7, C-4  
LBE, 3-7, B-4, C-4  
LBT, 3-6, B-3, C-4  
LBX, 3-12, B-7, C-4  
LBY, 3-12, B-7, C-4  
LDA, 3-2, B-2, C-4  
LDB, 3-2, B-2, C-4  
LDX, 3-12, B-7, C-4  
LDY, 3-12, B-7, C-4  
length attribute, 4-60  
LFA, 3-20, 3-23, B-12, C-4  
LFB, 3-20, 3-23, B-12, C-4  
LIA, 3-16, B-8, C-4  
LIAC, B-8, C-4  
LIB, 3-16, B-8, C-4  
LIBC, B-8, C-4  
LIST, 4-25, 4-28, B-15, C-4  
list file, E-5  
list output, 1-7, E-2  
listing control instructions, 1-14, 4-1, 4-25, B-1, B-15  
LIT, 2-9, 4-37, 4-42, B-16, C-4  
literal values, 2-9, 4-42, 4-43

literals,  
   =B, 2-9  
   =D, 2-9  
   =F, 2-9  
   =L, 2-9  
   =R, 2-9  
   =S, 2-9  
 LITF, 2-9, 4-37, 4-43, B-16, C-4  
 loader and generation control instructions, B-1, B-14  
 loader control instructions, 4-1, 4-17  
 local assembly-time variable, 4-56  
 LOD, 4-17, B-14, C-4  
 logical negation, 4-59  
 logical operators, 4-59, 4-66, 4-68  
 LSHIFT (system macro), K-2, K-23  
 LSL, 3-14, B-10, C-4  
 LSR, 3-14, B-10, C-4  
 LST, C-4, L-13

## M

machine instructions, 1-11, 2-4, 3-1, B-1, B-2  
 MACLIB statement, 4-13, 5-4, 5-19, B-18, C-4  
 Macro Assembler operations, E-1  
 macro body, 5-8  
 macro call statement, 1-1, 1-11, 5-1, 5-2, 5-3  
 MACRO control statement, E-2  
 macro definition, 1-1, 1-11, 2-4, 5-1, B-1, B-18  
 Macro ENDMAC statement, 5-2, 5-10  
 macro example, K-3  
 macro libraries, 5-1, 5-4  
 macro name statement, 5-2, 5-6  
 macro parameters, 4-54, 5-1, 5-6, 5-11  
 MACRO statement, 5-2, 5-5, B-18, C-4  
 MAX (system macro), K-2, K-15  
 MBF, 3-20, B-13, C-4  
 MBI, 3-20, B-13, C-4  
 MBT, 3-5, B-3, C-4  
 MBW, 3-20, B-13, C-4  
 memory protect fence, 3-23  
 memory reference instructions, 3-1, 3-2, B-1, B-2  
 memory spaces, 4-2  
 MESSAGE (system macro), K-2, K-25  
 messages during assembly, E-1, E-11  
 MIA, 3-16, B-8, C-4  
 MIAC, B-8, C-4  
 MIB, 3-16, B-8, C-4  
 MIBC, B-8, C-4  
 MIC, C-4, L-14  
 microcode replacements, 1-4, 4-23  
 microcoding capabilities, 1-4



MIN (system macro), K-2, K-16  
MNOTE, 4-68, 4-74  
MPY, 3-13, B-10, C-4  
MSEG, 4-35, 4-36, B-16, C-4  
multiple modules, 1-14, 4-13  
multiplication, 2-12  
MVW, 3-5, B-3, C-4  
MWF, 3-20, B-13, C-4  
MWI, 3-20, B-13, C-4  
MWW, 3-20, B-13, C-4

## N

NAM, C-5  
NAM statement, 1-4, 4-4, 4-13, B-14  
negate operator, 4-60  
nesting of macro definitions, 5-1, 5-16  
no-operation, B-3  
no-operation instruction, 3-1, 3-13, B-1, B-3  
non-extended relocatable records, 1-2, 1-3  
NOP, 3-13, B-3, C-5  
numeric terms, 2-7

## O

OCT, 4-37, 4-44, B-16, C-5  
octal constants, 4-39, 4-44  
old literal constructs, L-2  
old pseudo ops, L-3  
OLDRE, 1-3, E-2  
on-line loading of Macro/1000, E-1, E-13  
opcode field, 2-1, 2-2, 2-4  
operand field, 2-1, 2-2, 2-4, 2-5  
operator precedence, 2-11  
options, E-7  
ORB, C-5, L-3  
ORG, 4-4, 4-9, B-14, C-5  
ORR, 4-4, 4-12, B-14, C-5, L-4  
OTA, 3-16, 3-23, B-8, C-5  
OTAC, B-8  
OTB, 3-16, 3-23, B-8, C-5  
OTBC, B-8, C-5  
overflow bit, 3-1, 3-15, 3-17, B-1, B-8

## P

PAA, 3-20, B-12, C-5  
PAB, 3-20, B-12, C-5  
PBA, B-12, C-5

Index-8

PBB, 3-21, B-12, C-5  
 PBS, 3-21  
 program linkage instructions, 4-1, 4-19, B-1, B-15  
 program location counter, 1-12, 2-7, 4-2  
 program relocatable space, 1-12, 4-2, 4-3, 4-11, 4-35  
 program relocation, 1-12  
 program types, 4-6  
 programming aids, 1-11  
 pseudo operations, 2-4, 4-1, B-1, B-14

## R

RAL, 3-7, B-4, C-5  
 RAM, C-5, L-15  
 RAR, 3-7, B-4, C-5  
 RBL, 3-7, B-4, C-5  
 RBR, 3-7, B-4, C-5  
 recursion, 5-17  
 redefinition of opcodes, 5-7  
 register reference instructions, 3-1, B-1  
 register reference, alter-skip group, 3-7, 3-10, B-1, B-5  
 register reference, alter/skip group, B-5  
 register reference, shift-rotate group, 3-7, B-1, B-4  
 register reference, shift/rotate group, B-4  
 RELOC, 1-12, 4-4, 4-11, 4-13, 4-22, 4-35, 4-47, B-14, C-5  
 relocatable assembly, E-2  
 relocatable code, 1-4  
 relocatable expressions, 2-11  
 relocatable program, 4-4  
 relocatable records, extended, 1-2, 1-3  
 relocatable records, non-extended, 1-2, 1-3  
 relocatable spaces, 1-12  
 REP, C-5, L-7  
 REPEAT, 4-68, 4-73, B-17, C-5  
 replacement formats, 3-25  
 RESOLVE (system macro), K-2, K-24  
 ROTATE (system macro), K-2, K-21  
 RPL, 4-19, 4-23, B-15, C-5  
 RRL, 3-14, B-10, C-5  
 RRR, 3-14, B-10, C-5  
 RSA, 3-21, B-13, C-5  
 RSB, 3-21, B-13, C-5  
 RSS, 3-10, B-5, C-5  
 run string parameters, E-4  
 RVA, 3-21, B-13, C-5  
 RVB, 3-21, B-13, C-5

SBT, 3-6, B-3, C-5  
SBX, 3-12, B-7, C-5  
SBY, 3-12, B-7, C-5  
selective assembly, E-2  
SETBIT (system macro), K-2, K-17  
SEZ, 3-10, B-5, C-5  
SFB, 3-6, B-3, C-5  
SFC, 3-16, B-8, C-6  
SFS, 3-16, B-8, C-6  
shift-rotate group, 3-7, B-4  
SJP, 3-22, B-12, C-6  
SJS, 3-22, B-12, C-6  
SKP, 4-25, 4-31, B-15, C-6  
SLA, 3-7, 3-10, B-4, B-5, C-6  
SLB, 3-7, 3-10, B-4, B-5, C-6  
SOC, 3-17, B-9, C-6  
SOCC, B-9, C-6  
software replacements, 3-24  
SOS, 3-17, B-9, C-6  
SOSC, B-9, C-6  
source file, 1-4, E-4  
source statements, 2-1  
SPC, 4-25, 4-32, B-15, C-6  
SSA, 3-10, B-5, C-6  
SSB, 3-10, B-5, C-6  
SSM, 3-22, B-12, C-6  
STA, 3-3, B-2, C-6  
statement continuation, 2-14  
statement length, 2-14  
STB, 3-3, B-2, C-6  
STC, 3-16, B-8, C-6  
STCC, B-8, C-6  
STF, 3-16, B-8, C-6  
STO, 3-17, B-9, C-6  
STOP (system macro), K-2, K-26  
storage allocation, 4-1, 4-35, B-1, B-16  
string comparison, 4-65  
STX, 3-12, B-7, C-6  
STY, 3-12, B-7, C-6  
SUBHEAD, 4-25, 4-27, B-15, C-6  
subheading, 4-27  
subroutine operations,  
    CALL, K-5  
    ENTRY, K-5  
    EXIT, K-5  
substring attribute, 4-62  
SUBTRACT (system macro), K-2, K-14  
SUP, 4-25, 4-33, 4-34, B-15, C-6  
SWP, 3-14, B-10, C-6  
SXB, 3-10  
SYA, 3-21, B-12, C-6  
SYB, 3-21, B-12, C-6

subroutine operations,  
     CALL, K-5  
     ENTRY, K-5  
     EXIT, K-5  
 substring attribute, 4-62  
 SUBTRACT (system macro), K-2, K-14  
 subtraction, 2-12  
 SUP, 4-25, 4-33, 4-34, B-15, C-6  
 SWP, 3-14, B-10, C-6  
 SXB, 3-10  
 SYA, 3-21, B-12, C-6  
 SYB, 3-21, B-12, C-6  
 symbol definition instructions, 4-1, 4-45, B-1, B-16  
 symbol table, 1-10, E-2  
 symbolic addressing, 1-11  
 symbolic terms, 2-5  
 system assembly-time variables, J-1  
 system macro library, K-1  
 system macros, descriptions of, K-5  
 SZA, 3-10, B-5, C-6  
 SZB, B-5, C-6

## T

TBS, 3-6, B-3, C-6  
 terms, 2-4, 2-5  
 TESTBIT (system macro), K-2, K-19  
 TEXT (system macro), K-2, K-25  
 text and message, text definition, K-25  
 TYPE (system macro), K-2, K-26  
 TYPE and STOP, communication with RTE, K-26  
 type attribute, 4-61

## U

UJP, 3-22, B-12, C-6  
 UJS, 3-22, B-12, C-6  
 unary operators, 2-11, 4-59, 4-68  
 UNL, C-6, L-13  
 UNS, 4-25, 4-33, 4-34, B-15, C-6  
 upper-case attribute, 4-62  
 USA, 3-21, B-12, C-6  
 usage of AWHILE, 4-72  
 USB, 3-21, B-12, C-6  
 using AIF and AELSEIF, 4-70  
 using macro libraries, 5-4

## W

WEXT, B-15  
what the system Macros do, K-1  
word processing, B-1  
word, byte and bit-processing, B-3  
word-processing, 3-1, 3-4, B-3  
work file, E-8  
writing macro definitions, 5-1, 5-5

## X

XAX, 3-11, B-7, C-7  
XAY, 3-11, B-7, C-7  
XBX, 3-11, B-7, C-7  
XBY, 3-11, B-7, C-7  
XCA, 3-22, B-13, C-7  
XCB, 3-22, B-13, C-7  
XIF, C-7, L-5  
XLA, 3-22, B-13, C-7  
XLB, 3-22, B-13, C-7  
XMA, 3-21, B-13, C-7  
XMB, 3-21, B-13, C-7  
XMM, 3-21, B-13, C-7  
XMS, 3-21, B-13, C-7  
XOR, 3-2, B-2, C-7  
XSA, 3-22, B-13, C-7  
XSB, 3-22, B-13, C-7

