HEWLETT PACKARD

Macro/1000

Reference Manual

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

Software Technology Division
11000 Wolfe Road
Cupertino, CA 95014-9804

Manual Part No. 92059-90001
E1292

Printed in U.S.A. December 1992
Third Edition

NOTICE

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE 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.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARs 252.227.7013.

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 Manual Numbering File or the Computer User's Documentation Index. (The Manual Numbering File is included with your software. It consists of an "M" followed by a five digit product number.)

First Edition	Oct 1981	
Update 1	Apr 1982	
Update 2	Jul 1982	
Reprint	Jul 1982	Updates 1 & 2 Incorporated
Update 3	Jun 1983	
Reprint	Jun 1983	Update 3 Incorporated
Update 4	Dec 1983	
Reprint	Dec 1983	Update 4 Incorporated
Update 5	Oct 1984	Manual Errata
Reprint	Oct 1984	Update 5 Incorporated
Update 6	Jan 1985	Update Operation, Installation
Reprint	Jan 1985	Update 6 Incorporated
Update 7	Jan 1986	Manual Enhancements
Reprint	Jan 1986	Update 7 Incorporated
Second Edition	Aug 1987	Rev. 5000 (Software Update 5.0)
Third Edition	Dec 1992	Rev. 6000 (Software Update 6.0)

Preface

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

The user of this manual is assumed to have an in-depth knowledge of assembler programming and of the HP 1000 E/F-, and A-Series machine instruction sets.

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 computer Operating and Reference Manual for your model computer which will explain the specific machine instructions available on that machine.

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

Chapter 5 Describes the Macro/1000 language and how to create and access macro definitions. Macro libraries are also discussed.

Also included are the following Appendices:

Appendix A	Assembler Error Messages
Appendix B	Macro/1000 Instruction Set
Appendix C	HP 1000 Computer Instruction Set (Octal Opcode)
Appendix D	HP 1000 Computer Base and Extended Instruction Sets
Appendix E	Macro/1000 Assembler Operations
Appendix F	Cross Reference Table Generator
Appendix G	HP Character Set
Appendix H	Relocatable Record Formats
Appendix I	Implementation Notes
Appendix J	Backward Compatible Constructs
Appendix K	System Assembly-Time Variables
Appendix L	HP 1000 Macro Library
Appendix M	CDS Assembly Language Programming
Appendix N	CDSONOFF Macro Library
Appendix O	Program Types

Table of Contents

Chapter 1
Introducing the Macro Assembler

Chapter 2
Coding Format

Chapter 3
Machine Instructions

Chapter 4
Assembler Instructions

Chapter 5
Using Macros

Appendix A
Assembler Error Messages

Appendix B
Macro/1000 Instruction Set

Appendix C
HP 1000 Computer Instruction Set (Octal Opcode)

Appendix D
HP 1000 Computer Base and Extended Instruction Sets

Appendix E
Macro/1000 Assembler Operations

Appendix F
Cross-Reference Table Generator

Appendix G
HP Character Set

Appendix H
Relocatable Record Formats

Appendix I
Implementation Notes

Appendix J
Backward Compatible Constructs

Appendix K
System Assembly-Time Variables

Appendix L
HP 1000 Macro Library

Appendix M
CDS Assembly Language Programming

Appendix N
CDSONOFF Macro Library

Appendix O
Program Types

List of Illustrations

Tables

1

Introducing the Macro Assembler

Macro/1000 permits you to use all supported machine instructions for HP 1000 Computers. The Macro/1000 Assembler (MACRO) translates symbolic source language into machine code for execution on the computer. The source language provides mnemonic operation codes (opcodes), assembler-directing pseudo operations, 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 disk file or an input device. The resultant relocatable or absolute object program is output to a disk 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/1000 acts differently depending on what you specify.

If you specify ASMB in the control statement, Macro/1000 behaves in the same manner as the ASMB Assembly Language. Macro defining abilities are available, but because the ASMB function of Macro/1000 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/1000 then behaves as shown in this manual. MACRO produces extended relocatable records. If you have code written in ASMB and wish to run Macro/1000 with MACRO specified in the control string, be aware that Macro/1000 reserves some characters for special purposes:

A	–	MACRO assigns to A the value 0 (A EQU O)
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 to, greater than, less than) used as comparison operators.
‘	–	(single quote) designates a character string
@	–	(at-sign) designates indirect addressing

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 J (Backward Compatible Constructs) of this manual explains these instructions.

Relocatability

MACRO 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/1000. 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. The use of these records with other loaders or older RTE generators will cause unpredictable results. OLDRE must be executed with the file containing extended records before they can be loaded by loaders that do not accept extended records, or generated using older RTE generators.

Machines with Microcoding Capabilities

Some HP 1000 machines have software equivalents for instructions that are implemented in microcode in others. For example, some floating point instructions are microcoded in some CPUs and not others.

The instruction “.PWR2” (power of 2) is not microcoded in the A600. Therefore, MACRO generates a JSB 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 JSB with a microcode instruction or a JSB to the software routine. Appendix C of this manual defines the instructions that have software equivalents on a particular machine. Refer to the Operating and Reference Manual for your machine for more information.

MACRO will replace an instruction with microcode if requested. The ‘I’ option in the control statement (discussed in Appendix E) causes MACRO to generate microcode replacements for these instructions. You may specify the ‘I’ option if your machine has microcoding. Note that, in general, the software is more flexible if the ‘I’ option is not used, since the loader can then tailor the module to fit the hardware at load time.

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 statements, depending on whether the module is relocatable or absolute:

Relocatable Control Statement	Absolute Control Statement
NAM statement	ORG statement
END statement	END statement

```
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

The control statement (MACRO,R,L,T) 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/ORG 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. The END statement, however, can be followed by conditional assembly or other statements that do not produce code. There can also be another module or NAM-END pair.

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

- expands macros,
- checks for syntax errors in the source statements,
- creates the list file, and
- creates relocatable code.

The relocatable code produced by MACRO is then ready to be loaded using the loader program, LINK. LINK produces memory image code that the computer can execute. The whole process is illustrated in Figure 1-2.

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

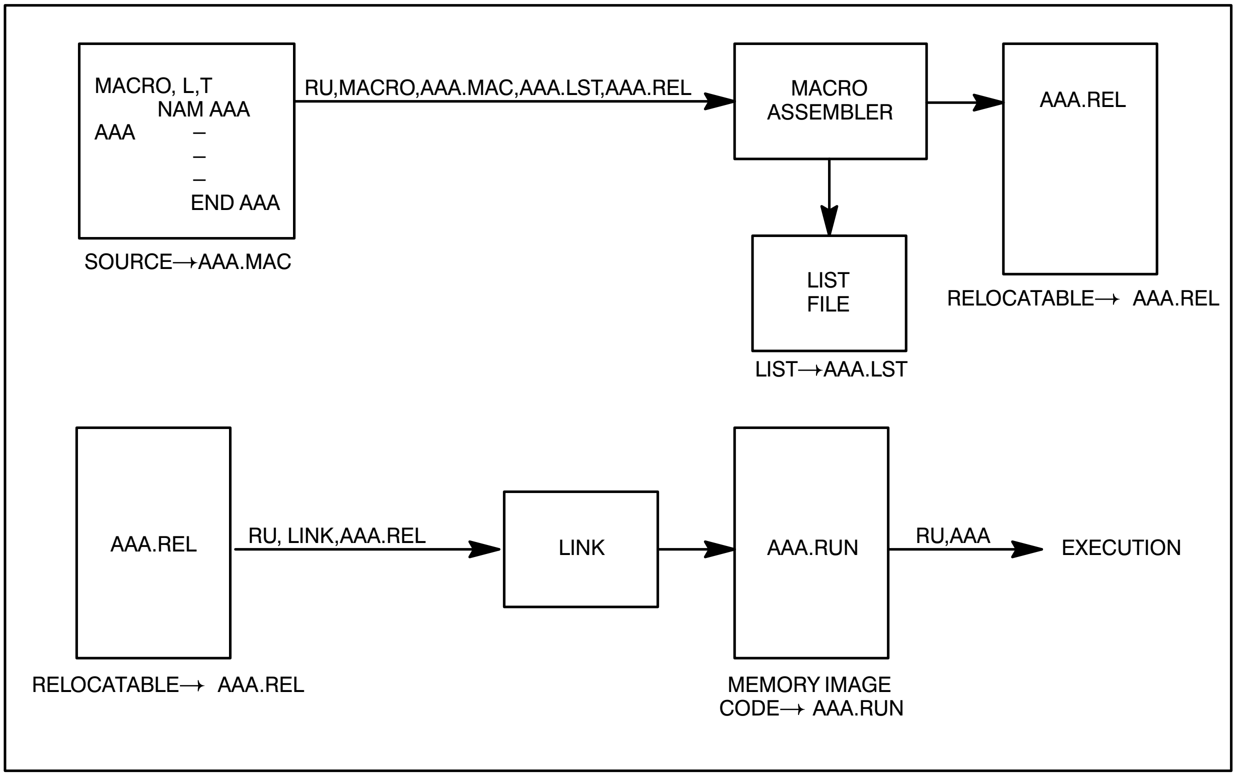


Figure 1-2. Assembly Process

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 symbol that indicates the type of relocation to be done for the operand field are as follows:

Symbol	Type of Relocation
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

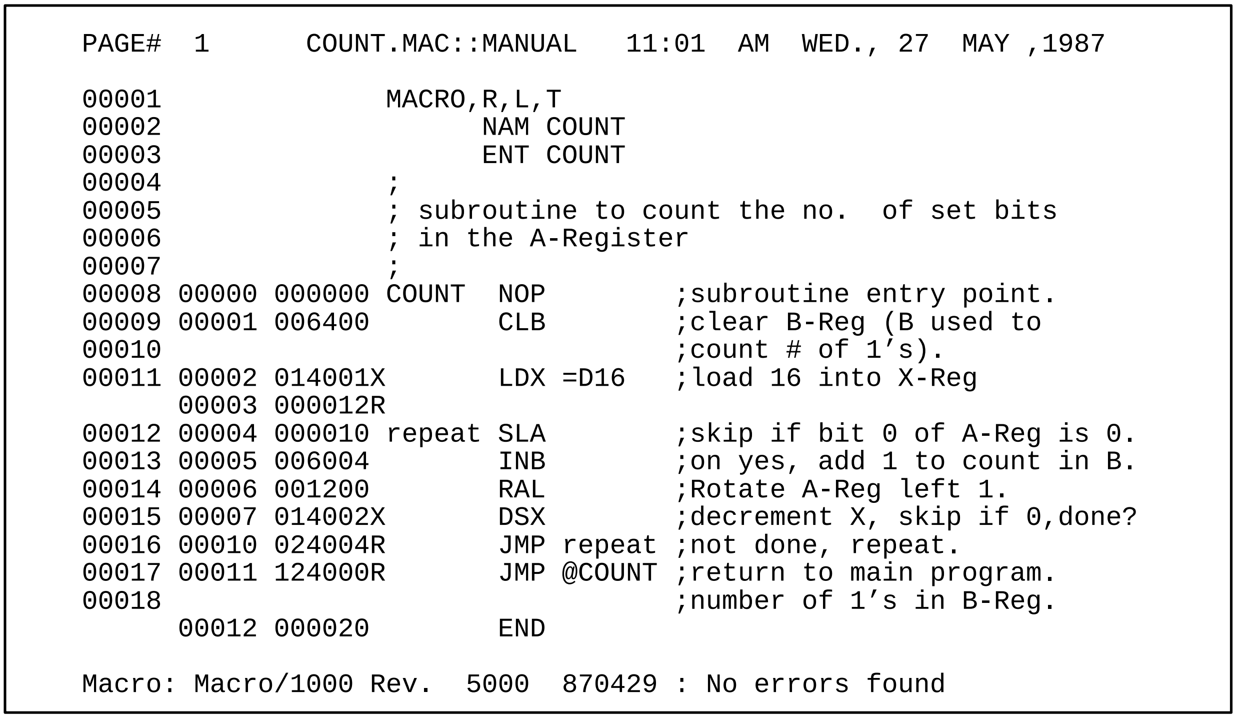
In a CDS environment (see Appendix M for details and refer to the appropriate programmer's reference manual to see if your machine has these features) the relocation type symbols are:

Blank	absolute
c	code
d	data
s	static
l	local
x	external
e	ema
C	Common

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 the first non-blank column 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.

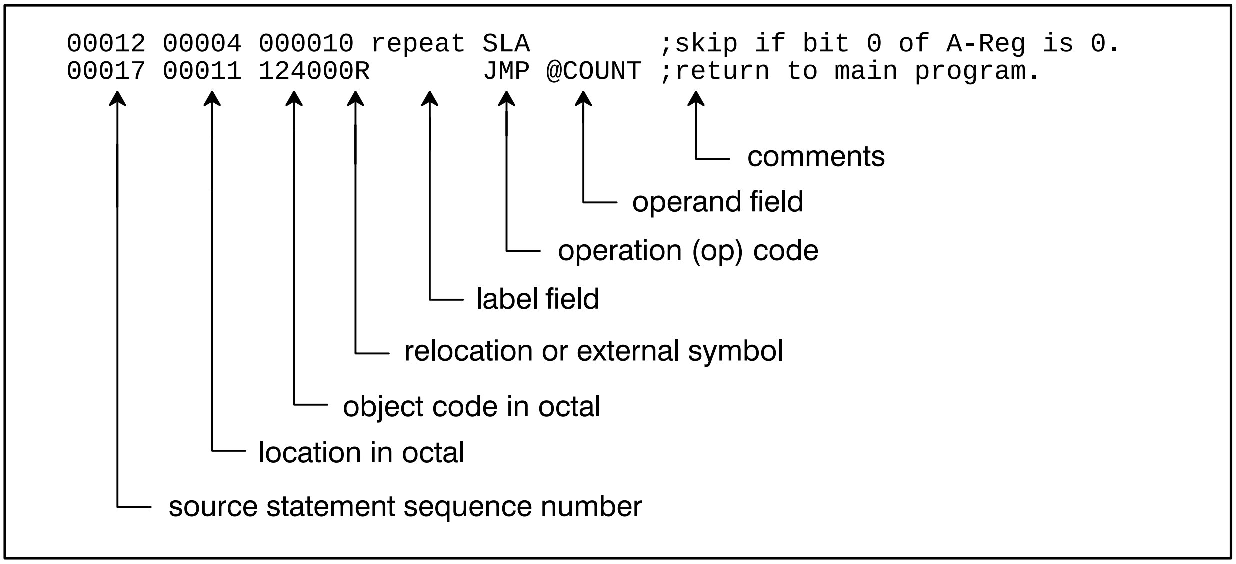


```
PAGE#  1      COUNT.MAC::MANUAL   11:01  AM  WED., 27  MAY ,1987

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: Macro/1000 Rev.  5000  870429 : No errors found
```

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.
```

Figure 1-4. Listing Fields

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. After the error list, the number and text of each error is repeated for each unique error number reported in the error list.

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

Note The symbols

```
*** RELOC **
*** ORG  ***
*** ORB ****
*** ORR ****
```

may appear in the symbol table output if statements by the same name appear in the source. These symbols are put in the symbol table to facilitate their appearance in the cross-reference and will always have the value 0. These symbols are not legal Macro/1000 symbols and do not conflict with any legal symbols.

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

MACRO language consists of the following opcodes:

- Machine instructions, which instruct the machine to do something such as manipulate registers or send flags to the operating system.
- Assembler instructions, which instruct MACRO to do something such as create space for a value or specify a listing option.
- 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, or
- 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 six different types of relocatable spaces:

- program relocatable,
- base page relocatable,
- EMA relocatable,
- SAVE relocatable,
- common relocatable, and
- labeled common relocatable.

In the CDS environment (see Appendix M for details and refer to the programmer's reference manual for your computer to see if your machine has these features) the available spaces are as follows:

- code
- data
- static
- local
- EMA
- common
- labeled 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 opcode at the start of the program. The NAM opcode causes the program location counter to be set to zero for a relocatable program, the ORG opcode 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 source file).

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 these 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 opcode, LIST, has a keyword parameter to do many of the above options. Another pseudo opcode, COL, controls the columns in which the mnemonic, operand, and comments start in the assembled listing.

2

Coding Format

The source code for an assembly language program consists of a series of source statements. The formats of the source statements are 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
```

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.

In addition to source code format, the following example illustrates parameter passing and indirect addressing. The calling sequence to this subroutine is JSB CONVT followed by DEF BUFF; where BUFF contains the ASCII value to convert. The binary value is returned to the calling program in the A-Register.

```
     JSB CONVT
     DEF BUFF
     STA BINVALUE
     :

MACRO,R,L
     NAM CONVT
     ENT CONVT
*
* Subroutine to convert a positive four digit number from its
* ASCII form into the corresponding binary value.  The A-Register
* contains the binary value on return to the calling program.
*
CONVT NOP          ; subroutine entry point
      LDA @CONVT   ; get the address of the ASCII value
      STA PNTR     ; save in PNTR
      ISZ CONVT    ; increment return address
      ISZ PNTR     ; set pointer to address of the second word
      LDA @PNTR    ; put this word in the A-Register
      AND MASK     ; mask out the upper four bits
      STA TOTAL    ; save the result in TOTAL
      LDA @PNTR    ; get the second word again
      AND TMASK    ; mask out the lower four bits
      ALF,ALF      ; rotate the A-Register
      MPY TEN      ; multiply by 10
      ADA TOTAL    ; add TOTAL to the A-Register
      STA TOTAL    ; save the result back into TOTAL
      LDA PNTR     ; get back the address of the second word
      ADA =D–1     ; decrement this address
      STA PNTR     ; store the address back into PNTR
      LDA @PNTR    ; put the first word in the A-Register
      AND MASK     ; mask out the upper four bits
      MPY HUND     ; multiply by 100
      ADA TOTAL    ; add TOTAL to the A-Register
      STA TOTAL    ; save the result back into TOTAL
      LDA @PNTR    ; get the first word again
      AND TMASK    ; mask out the lower four bits
      ALF,ALF      ; rotate the A-Register
      MPY THOU     ; multiply by 1000
      ADA TOTAL    ; add TOTAL to the A-Register
      JMP @CONVT   ; return to the calling program
*
* Local constants and Variables
*
TEN   DEC 10
HUND  DEC 100
THOU  DEC 1000
MASK  OCT 17
TMASK OCT 7400
PNTR  BSS 1
TOTAL BSS 1
      END
```

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 uppercase
!	–	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 a starting character

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 Computer 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.
```

Discretion should be used in the placement of a comma in the operand field. A blank space is treated as the beginning of a comment field unless it is preceded by a comma. The use of a semicolon as a comment delimiter, as shown above, may or may not be required. (See the section titled “Comment Field” in this chapter for more information.)

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 opcode.

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
```

Numeric Terms

A numeric term may be a decimal or octal integer. A decimal number is represented by one to ten digits within the range -2,147,483,648 to 2,147,483,647. An octal number is represented by one to eleven digits followed by the letter B (0 to 37777777777B).

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
DMX DDEF 2147483647  ; maximum positive double integer
```

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
```

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 – lowercase mapped to upper), and digits.

By convention, system assembly-time variables begin with the character sequence "&.". (Refer to Appendix K for more information on system assembly-time variables.) The assembler will not mark an error if you declare user assembly-time variables starting with "&.". However, if you declare a variable that is the same as a system variable, an error will result. To ensure 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:

=A A two ASCII-character string, no quotes.

=B An octal integer, one to six digits between 0 and 7, resulting in an octal value between 0 and 177777B.

=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}.

=J Same as =L except that it generates a 32-bit result.

=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. The one exception is when the literal appears in the opcode of a memory reference instruction in CDS code space. (See the subsection in this chapter titled “Legal Uses of Expressions”.)

=R A right-justified, zero-filled ASCII character.

=S A string surrounded by single quotes.

If you use the same literal in more than one instruction or if different literals have the same value (for example, =B100 and =D64), only one value is generated, and all instructions using these literals refer to the same location. Literals can be specified only in the following memory reference, register reference, EAU, and pseudo opcodes:

ADA	CPB	LDX	
ADB	DDEF	LDY	
ADX	DIV	MBT	This group can use:
ADY	IOR	MPY	=A =B =D =L =R =S,
AND	JRS	MVW	however, =S must be 1 or 2 characters.
CBS	LDA	SBS	
CBT	LDB	TBS	
CMW		XOR	
CPA			
DLD	FDV	FSB	This group can use:
FMP	FAD		=F =J =S with 3 or 4 characters.
DEF	DBL	DBR	This group can use any literal.

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  =L(ARRAY)       ; Load A-Register with the address of
                          ; ARRAY.
                          ; NOTE: Parentheses are for clarity, they
                          ;       are not required.
     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.
     LDA  =RA             ; The lower byte of the A-Register
                          ; contains 101 octal – the ASCII value
                          ; of "A"; the upper byte contains zeros.
```

Literals in CDS

Literals referenced by:

```
      LDA
      LDB
      IOR
      XOR
      AND
      CPA
      CPB
      ADA
      ADB
      DLD
```

instructions in CODE space (that is, after a RELOC CODE) generate special relocatable record entries. These entries cause the loader to allocate the value required in a link area in code space where they can be directly accessed by the instruction. DLD is actually converted to LDA, LDB by the loader. The listing for these references will show an address of 0 and will not appear in the literal pool (unless they are also referenced by some other instruction).

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 operators	-	(negate)
	:SY:	(symbol ID)
	:MR:	(memory relocatability)
	:ICH:	(integer value of a character)
arithmetic operators	*	(multiply)
	/	(divide)
	+	(add)
	-	(subtract)

The unary operator :SY: returns, as an absolute value, the symbol ID of the expression it operates on. This is the same number as in the external and allocate records for the symbol. If the expression does not reference an external or allocate symbol, 0 is returned.

The unary operator :MR: returns, as an absolute value, the relocatability of its operand as follows:

0 = Absolute
1 = Program relocatable
2 = Base page relocatable
3 = Common relocatable
4 = Pure code relocatable (CDS only)
5 = EMA relocatable
6 = SAVE relocatable
7 = External
9 = Allocate EMA
10 = Allocate SAVE
12 = Allocate common
20 = Two or more of the above

Operator Precedence

Conventionally, expressions are evaluated from left to right in the statement. However, 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

An expression may have relocatability in the following spaces:

Program
Base page
Common
Code
Local EMA
Save
One external space

Each of these relocatabilities may be multiple and either positive or negative. For example:

Given the following symbols:

```
     RELOC PROGRAM

PROG EQU *
w    EQU *+10
     RELOC BASE
BASE EQU *
     RELOC SAVE
SAVE EQU *
EXT1 ALLOC COMMON, 20
     EXT EXT2
```

Then:

```
PROG + BASE – SAVE
```

has: +1 relocatability in the program and base page spaces, and −1 relocatability in the save space.

```
EXT1 + EXT2
```

is illegal because it has relocatability in more than one external space (ALLOCs are external).

```
PROG + PROG + EXT1
```

has +2 relocatability in the program space and +1 in the external space EXT1.

Any legal expression may be used in an EQU pseudo opcode. However, use of expressions (or EQUate symbols) with relocability in more than one space is restricted as follows:

1. The expressions for

   ```
   DEF
   DDEF
   =L
   =J
   ```

   may have relocatability in 0 or 1 spaces and that relocatability must be in the range −8 through +7. Therefore, it follows that:

   ```
   DEF    w+w     (+2)   is legal.  (an example of a byte address)
   DEF    -w      (−1)   is the negative of w.
   DEF    -w-w    (−2)   is the negative byte address of w.
   DDEF   w+w     (+2)   EMA byte address.
   ```

2. The opcodes DBL and DBR imply relocatability (of +2) is to be imposed on their expressions. These expressions must have relocatability in 0 or 1 spaces and the relocatability value must be −1 or 1. For example,

   ```
    DBL -w
   ```

   is the negative of the byte address of the left-hand byte of word w.

No other opcodes may have expressions with relocatability in more than one space and that relocatability must be 1. Absolute expressions must not have relocatability in any space. For example,

```
LDA   w+w
```

is illegal because it has +2 relocatability.

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 nonblank 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

A blank space in the operand field is also treated as a comment delimiter unless the blank space is preceded by a comma. Note the following distinction in the placement of a comma and blank space:

```
    EXT   sym1,  sym2          sym2 is treated as a second operand in this line

    EXT   sym1  ,sym2          sym2 is treated as a comment in this line
```

An asterisk (*) appearing as the first character on a line also denotes the entire line as a comment.

Within a macro definition, lines beginning with .* are treated as comments and are not expanded with the rest of the macro when it is called. Also refer to Chapter 5.

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 HP 1000 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 opcode. 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    ; into 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 an indirect reference to a program link location (established by the loader in either the base page or the current page) and then stores a direct 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
```

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 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 and throughout 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.

Where operands are shown stacked vertically, only one operand may be used.

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.

The first form is:

$$[\textit{lable}]\ \ \textit{opcode}\left\{\begin{array}{l}\textit{m}\\ \textit{@m}\\ \textit{literal}\end{array}\right\}[\texttt{;}\textit{comments}]$$

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.

The second form is:

$$[\textit{lable}]\ \ \textit{opcode}\left\{\begin{array}{l}\textit{m}\\ \textit{@m}\end{array}\right\}[\texttt{;}\textit{comments}]$$

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

Note Instructions in this group are implemented by calls to external subroutines unless the ‘I’ option is used in the Macro statement. Refer to Appendix E for details on the ‘I’ option.

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 address occupies 16 bits: bits 1-15 indicate the address of the word containing the byte, and bit 0 indicates a high order byte (bit is clear) or low order byte (bit is set). The byte-processing instructions are LBT, SBT, MBT, CBT, and SFB.

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.

The word, bytes, and bit processing instructions can take one of three syntactical forms.

The first of these forms is:

$$[\mathit{lable}]\ \ \mathit{opcode} \left\{ \begin{array}{l} m \\ @m \\ \mathit{literal} \end{array} \right\} [\texttt{;}\mathit{comments}]$$

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.

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 the 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 word 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 the 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 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 opcodes DBL and DBR in Chapter 4 for more information on byte addressing.

The second syntactical form is:

[*lable*] *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:

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

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 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 next instruction is skipped.

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.

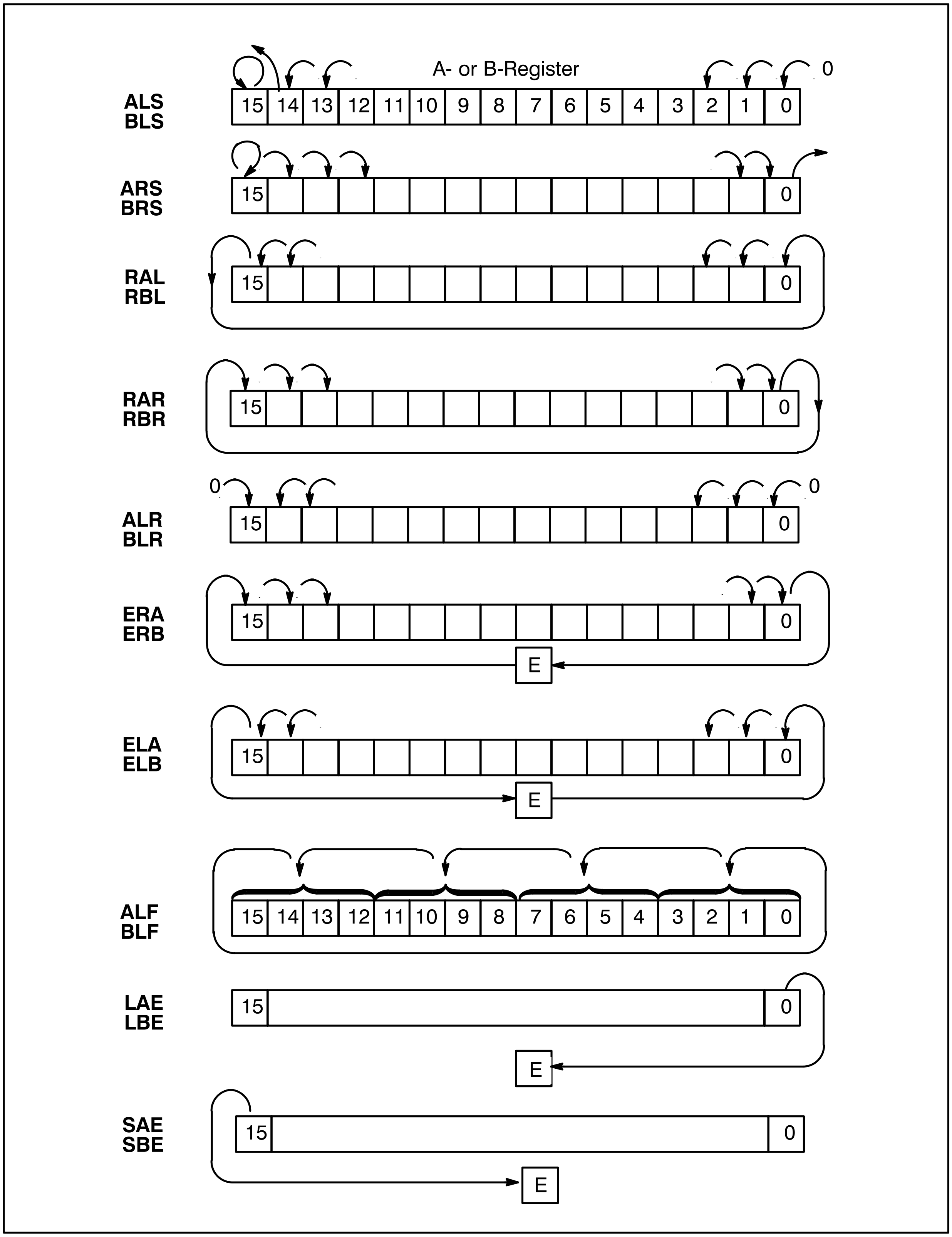


Figure 3-1. Instructions of the Shift-Rotate Group

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

```
          ⎡⎧ ALS ⎫⎤                  ⎡⎧ ,ALS ⎫⎤
[label]   ⎢⎪ ARS ⎪⎥ [,CLE] [,SLA]    ⎢⎪ ,ARS ⎪⎥   [;comments]
          ⎢⎪ RAL ⎪⎥                  ⎢⎪ ,RAL ⎪⎥
          ⎢⎪ RAR ⎪⎥                  ⎢⎪ ,RAR ⎪⎥
          ⎢⎨ ALR ⎬⎥                  ⎢⎨ ,ALR ⎬⎥
          ⎢⎪ ALF ⎪⎥                  ⎢⎪ ,ALF ⎪⎥
          ⎢⎪ ERA ⎪⎥                  ⎢⎪ ,ERA ⎪⎥
          ⎢⎪ ELA ⎪⎥                  ⎢⎪ ,ELA ⎪⎥
          ⎢⎪ SAE ⎪⎥                  ⎢⎪ ,SAE ⎪⎥
          ⎣⎩ LAE ⎭⎦                  ⎣⎩ ,LAE ⎭⎦
```

```
          ⎡⎧ BLS ⎫⎤                  ⎡⎧ ,BLS ⎫⎤
[label]   ⎢⎪ BRS ⎪⎥ [,CLE] [,SLB]    ⎢⎪ ,BRS ⎪⎥   [;comments]
          ⎢⎪ RBL ⎪⎥                  ⎢⎪ ,RBL ⎪⎥
          ⎢⎪ RBR ⎪⎥                  ⎢⎪ ,RBR ⎪⎥
          ⎢⎨ BLR ⎬⎥                  ⎢⎨ ,BLR ⎬⎥
          ⎢⎪ BLF ⎪⎥                  ⎢⎪ ,BLF ⎪⎥
          ⎢⎪ ERB ⎪⎥                  ⎢⎪ ,ERB ⎪⎥
          ⎢⎪ ELB ⎪⎥                  ⎢⎪ ,ELB ⎪⎥
          ⎢⎪ SBE ⎪⎥                  ⎢⎪ ,SBE ⎪⎥
          ⎣⎩ LBE ⎭⎦                  ⎣⎩ ,LBE ⎭⎦
```

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:

$$\left[\begin{Bmatrix}\texttt{CLA}\\ \texttt{CMA}\\ \texttt{CCA}\end{Bmatrix}\right] \quad \texttt{[,SEZ]} \quad \left[\begin{Bmatrix}\texttt{,CLE}\\ \texttt{,CME}\\ \texttt{,CCE}\end{Bmatrix}\right] \quad \texttt{[,SSA] [,SLA] [,INA] [,SZA] [,RSS]}$$

$$\left[\begin{Bmatrix}\texttt{CLB}\\ \texttt{CMB}\\ \texttt{CCB}\end{Bmatrix}\right] \quad \texttt{[,SEZ]} \quad \left[\begin{Bmatrix}\texttt{,CLE}\\ \texttt{,CME}\\ \texttt{,CCE}\end{Bmatrix}\right] \quad \texttt{[,SSB] [,SLB] [,INB] [,SZB] [,RSS]}$$

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

The first form 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 – Exchange B and X.
XBY – Exchange B and Y.

The second form is:

$$[\textit{lable}]\ \ \textit{opcode} \left\{ \begin{matrix} \textit{m} \\ \textit{@m} \\ \textit{literal} \end{matrix} \right\} [\texttt{;}\textit{comments}]$$

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.

The third form is:

$$[\textit{lable}]\ \ \textit{opcode} \left\{ \begin{matrix} \textit{m} \\ \textit{@m} \end{matrix} \right\} [\texttt{;}\textit{comments}]$$

Opcodes that require this form are:

JLA – Jump and load A.
JLB – Jump and load B.
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 fourth statement form using index register opcodes is:

[*label*] *opcode* *m* [;*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. The contents of the A- and B-Registers must be swapped if the results of the instruction (for example, MPY) are to be subsquently used in a double integer instruction. The A990 computer is an exception because it has a double integer arithemetic group of instructions (for example, MPYD, DIVD, etc.).

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

The first form is:

$$[\textit{lable}]\ \ \textit{opcode} \begin{Bmatrix} m \\ @m \\ \textit{literal} \end{Bmatrix} [\texttt{;}\textit{comments}]$$

Opcodes that require this form are:

DIV – Divide the 32-bit integer contents of B (high-order bits) and A (low-order bits) 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.

DIVD – Same as DIV, except that A contains the high-order bits and B contains the low-order bits.

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. The result is a 32-bit integer, with the high-order bits in the B-Register and the low-order bits in the A-Register.

MPYD – Same as MPY, except that the high-order bits of the result are in the A-Register and the low-order bits are in the B-Register.

The second form is:

$$[\textit{lable}]\ \ \texttt{DST} \begin{Bmatrix} m \\ @m \end{Bmatrix} [\texttt{;}\textit{comments}]$$

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

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

The third form is:

$$[\textit{label}]\ \ \textit{opcode}\ \ n\ \ [\texttt{;}\textit{comments}]$$

The six extended arithmetic register reference instructions provide shifting operations on the combined contents of B- and A-Registers. The B-Register contains the left (most-significant) bits, and the A-Register contains the right (least-significant) bits.

In these instructions, the range of *n* is from 1 to 16 bits.

Opcodes that require this form are:

ASL – Arithmetically shift B and A left *n* bits. The sign bit (bit 15 of B) is unaltered. Least significant bits are zeroed.
ASR – Arithmetically shift B and A right *n* bits. The sign bit (bit 15 of B) is extended.

LSR – Logically shift B and A right *n* bits. The most significant bits are zeroed.
LSL – Logically shift B and A left *n* bits. The least significant bits are zeroed.

RRL – Rotate B and A left *n* bits.
RRR – Rotate B and A right *n* bits.

The last form of the Extended Arithmetic Group is:

`SWP [;`*comments*`]`

This instruction exchanges the contents of 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 opcode. In such a case, the entry point referred to by the pseudo operation 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 clears 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
LIA
LIB
MIA
MIB
OTA
OTB
CLO
STC
SOC
SOS
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.

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.

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 two syntactical forms.

The first form is:

$$[\textit{lable}]\ \ \textit{opcode} \left\{ \begin{array}{l} \textit{m} \\ @\textit{m} \\ =\mathrm{F}\textit{n} \end{array} \right\}\ \ [;\textit{comments}]$$

Opcodes that require 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 the address specified by the operand and its following location or by the quantity defined by the literal from the two-word floating-point quantity defined in registers A and B. The result is stored in A and B.

The second form is:

$$[\textit{label}]\ \ \textit{opcode}\ \ [;\textit{comments}]$$

Opcodes 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 is an E- or F-Series and 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 one of three syntactical forms.

The first form is:

$$[\textit{lable}]\ \ \textit{opcode} \begin{Bmatrix} m \\ @m \\ \textit{literal} \end{Bmatrix} \begin{Bmatrix} m \\ @m \end{Bmatrix}\ \ [\texttt{;}\textit{comments}]$$

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.

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 words 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.

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 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.

The last form is:

$$[\mathit{lable}]\ \ \mathit{opcode} \left\{ \begin{array}{l} m \\ @m \end{array} \right\} [\texttt{;}\mathit{comments}]$$

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.

CDS Opcodes

The instructions in this group are available only on machines that support code and data separation (CDS). Consult the appropriate hardware and programmer's reference manuals to see if you can use these features. Appendix M of this manual explains the CDS environment in more detail.

PCAL – Call a subroutine.

HP 1000 A- and E/F-Series Replacements

The following list represents the instructions that are implemented by calls to external subroutines (unless the 'I' option is used in the Macro statement, see Appendix E).

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

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

4

Assembler Instructions

This chapter describes Assembler instructions, also known as pseudo operations or pseudo opcodes. The term “pseudo” 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-bit 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 six memory spaces are:

program relocatable
base page relocatable
EMA relocatable
SAVE relocatable
common relocatable
labeled common

A labeled common space is maintained for each labeled common referenced by the module.

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

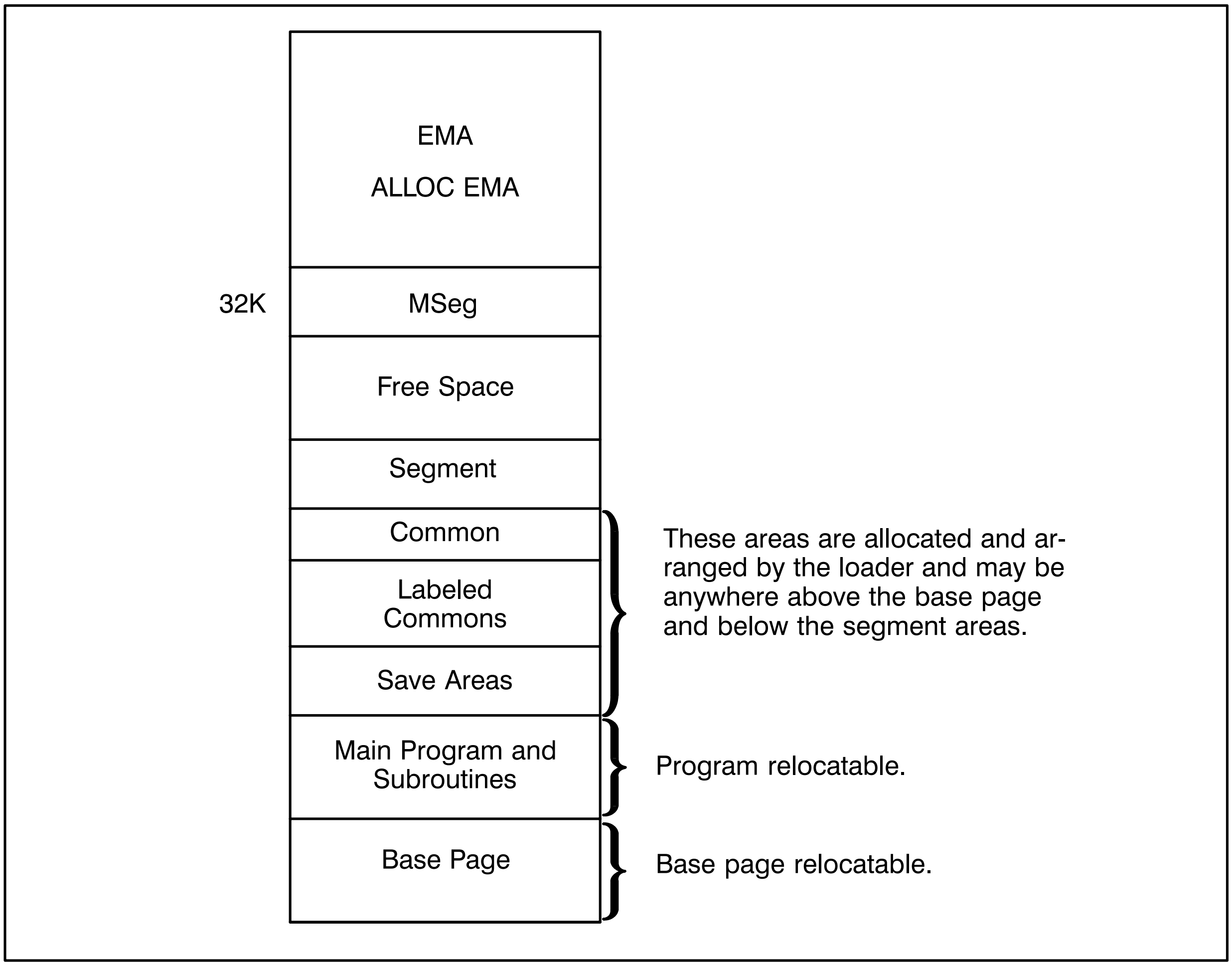

EMA
ALLOC EMA
32K
MSeg
Free Space
Segment
Common
Labeled Commons
Save Areas
Main Program and Subroutines
Base Page
These areas are allocated and arranged by the loader and may be anywhere above the base page and below the segment areas.
Program relocatable.
Base page relocatable.

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

Refer to “Program Relocation and Relocatable Spaces” in Chapter 1 for more information about these memory spaces.

In a CDS environment (consult the appropriate programmer’s reference manual), the memory spaces are:

code
data
static
local
EMA
common
labeled common

Appendix M outlines the user’s logical memory map for CDS programming.

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 overlayed 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 labeled common space holds variables declared to be in that labeled common. Each labeled common has its own location counter.

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*]]]]]]] [*comments*]

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).

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, or a comma must be used as a placeholder.

where:

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

type program type 1-8, 13, 14, 15, 30, 512 (refer to Appendix O for a definition of program types).

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 specifies the time the program will first run.

comments comments that will appear in the NAM record in the relocatable file. The comments must be preceded by a space.

Note

The five instructions SKP, HED, SUBHEAD, NAM, or after an END all cause the next line output to the list file to be preceded by:

1. A form feed: one in column one,
2. A page head, and
3. The currently active HED and SUBHEAD lines (if any).

The SPC instruction may also force this condition.

The NAM statement is always listed but SKP, HED, and SUBHEAD statements are only listed if they are in error.

This means that HED and SUBHEAD lines may be put above the NAM by entering them prior to NAM in the source file.

For example:

```
  :                                  :
  :                                 END
 END                                HED ...
                                    SUBHEAD ...
 NAM ...                            NAM ...
 HED ...                             :
 SUBHEAD ...                         :
  :
```

will put the NAM statement on a page by itself with no HED or SUBHEAD.

will put the NAM under the HED and SUBHEAD and continue the program on the same page.

Because no action is actually taken until a line is sent to the LIST file, the order of instructions is irrelevant. For example, in the absence of listing lines, the order of instructions in the following three groups is irrelevant:

```
HED                                 SKP
SUBHEAD         SUBHEAD             HED
SKP             HED                 SUBHEAD
```

Comments after an END and before a following NAM statement will appear with the module containing the END statement.

A typical NAM statement will look like this:

```
NAM XYZ,3
```

Example: The following is one way to time-schedule 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 disk resident) with priority 50.

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

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/1000 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 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 named PROG to be type 3 (background disk-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 LINK.

source statement:

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

loader listing:

```
CI> LINK,PROG.REL

PROG 40012 3994. 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.

Each time a module using one of these ATVs 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
```

ORG

`ORG` *operand* `[;`*comments*`]`

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 the 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, or
- 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 should be used only to specify non-code generation constructs (for example, BSS or EQU, etc.).

ORG to ALLOC symbols will take the symbol value of 0 rather than the previous high water mark (highest address), that is:

```
AB ALLOC     common,40
   ORG       AB
   ORG       AB + 20
```

ORGs to ALLOC common allow the common to be initialized.

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

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

CDS

`CDS` *keyword* `[;`*comments*`]`

The CDS statement instructs the assembler that your programs will be executing in a CDS (code and data separation) environment. This environment is described in Appendix M.

The keywords are:

`ON` turn on the CDS features of the assembler and prepare code to execute in the CDS environment.

`OFF` is included here for completeness. Omitting the CDS statement produces the same result.

RELOC

`RELOC` *keyword* `[;`*comments*`]`

The RELOC statement allows you to designate the relocation space in which the statements code or data is to be assembled. In a particular relocation space the initial RELOC starts at zero, while subsequent RELOC statements (to the same space) pickup at the address where its predecessor completed. ORG's into or within a relocation space go to the designated address.

In response to each RELOC, ORG, ORB, and at an END statement, two things happen:

1. The relocation address is checked to see if it is the highest yet received (the high water mark) for that relocation space. If it is at the high water mark, it is saved for comparison with subsequent RELOCs to that space.

2. The relocation space is checked to ascertain if the space used exceeds the space available. For ALLOC space this is the size declared, while for program space it is 32768.

Non-CDS Environment

In the non-CDS programming environment, the operand can be one of the following six keywords. Only the first three characters of the keyword are required:

`ALLOC`,*name* Assembles the following code or data in the ALLOC space. This code (or data) was previously defined with *name* as its label.

This is how ALLOC common is initialized in Macro/1000. Note that certain loaders (for example, the RTE-6/VM generator) require that such modules be of the type 512 plus the standard type (usually 7). When this is the case, the NAM record should be:

```
NAM K,519 ;set 512 + 7 type.
```

Any ALLOC space can be RELOCed into and any code can be generated in this space. The loader will not handle anything other than absolute data in the EMA space.

`BASE` Assembles the following data onto the base page.

`COMMON` Assembles the following data into the unnamed common space.

`EMA` Reserves the space that follows RELOC instruction in EMA. This is a special case of the RELOC command. Any labels in this space are local names used to refer to local EMA via DDEF statements since they represent 32-bit values.

`PROG` Assembles the following code or data into the program relocation space.

`SAVE` Assembles the following data into the SAVE relocation space.

If no RELOC space is specified, the assembler assembles the code or data into the program relocation 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 COMMON relocation space, no code generation or initialization is permitted:

```
      NAM  FOO
       :
      RELOC COMMON
      LDA   A           ; illegal.  Generates code.
      DEF   *           ; illegal.
      ABS   2           ; illegal.
       :
 FOO  BSS   10          ; legal.  No code generated.
```

This error may be detected at load time.

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
```

CDS Environment

In a CDS environment, the operand can be, in addition to PROG, COMMON, BASE, SAVE, EMA, one of the following four keywords:

`CODE` Assembles the following instructions into code space. This area is reserved for executable instructions.

`DATA` Assembles the following data declaration statements (for example, DEC, ASC, BSS) into the data area.

`LOCAL` Assembles the following space allocation statements (BSS only) into the local area. This area holds the values of variables that are local to a subroutine. They are active and accessible only when that subroutine has been called. The values cannot be initialized (such as with the DEC or ASC statement) and are of undefined value when the subroutine is entered. Local space is allocated off the stack on subroutine entry.

`STATIC` Assembles the ensuing data declaration statements (for example, DEC, ASC, BSS) into the STATIC data area. A static variable is local to a subroutine, but, unlike LOCAL area values, its value is preserved from one call to the next. This area is analogous to the SAVE area in a non-CDS environment.

Refer to Appendix M for examples of the use of each of these areas.

For example:

```
     RELOC COMMON
X    BSS   10
```

is equivalent to the FORTRAN statements:

```
COMMON  X(10)
INTEGER X
```

that is, local, blank common, and CDS usage.

ORR

```
ORR [;comments]
```

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 [operand][;comments]
```

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, this is indicated by the EOF (end of file).

The operand field contains a name appearing as a statement label in the current program, or it may be blank, if, however, the pending module is the main program, the operand field must be specified. The name identifies the transfer address (where program execution is to begin). In general, only one module in a program or overlay segment should contain a transfer address.

Note The five instructions SKP, HED, SUBHEAD, NAM, or after an END all cause the next line output to the list file to be preceded by:

1. A form feed: one in column one,
2. A page head, and
3. The currently active HED and SUBHEAD lines (if any).

The SPC instruction may also force this condition.

The NAM statement is always listed but SKP, HED, and SUBHEAD statements are only listed if they are in error.

This means that HED and SUBHEAD lines may be put above the NAM by entering them prior to NAM in the source file.

For example:

```
  :
  :
END

NAM ...
HED ...
SUBHEAD ...
  :
```

will put the NAM statement on a page by itself with no HED or SUBHEAD.

```
  :
END
HED ...
SUBHEAD ...
NAM ...
  :
  :
```

will put the NAM under the HED and SUBHEAD and continue the program on the same page.

Because no action is actually taken until a line is sent to the LIST file, the order of instructions is irrelevant. For example, in the absence of listing lines, the order of instructions in the following three groups is irrelevant:

```
HED                                   SKP
SUBHEAD          SUBHEAD              HED
SKP              HED                  SUBHEAD
```

Comments after an END and before a following NAM statement will appear with the module containing the END statement.

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 in one file and assembled together. Only one control statement is allowed per file. When a NAM statement is encountered, Macro/1000 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 contained in one module.

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 filedescriptor
```

The INCLUDE pseudo opcode 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 uppercase.

The following description of INCLUDE applies to all cases except that of macro library building. For information on INCLUDE in a macro library environment, refer to the "Creating Macro Libraries" section in Chapter 5.

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 opcodes. 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.

If the filedescriptor does not specify a path, MACRO will search the directory that contains the source file.

The comment field is not allowed.

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 opcodes, 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 Programmer's Reference Manual or Loader Manual for further detailed information.

LOD

`LOD` *n*`,`*string* `[;`*comments*`]`

The LOD statement is an instruction to the loader.

When some loaders encounter a LOD statement, they perform the function defined by the operand field.

The operand has two parts:

n – an expression which defines the number of words in the character string (two characters per word).

string – any legal loader command allowed before the SEarch or RElocate commands. (Refer to your Loader Reference Manual for details.)

Example:

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

Note If the length expression *n* is not a simple ATV or number, then string substitution is done on the whole LOD statement including the comments. In general, errors are ignored; however, MACRO insists on matched quotes and will remove unmatched quotes from quoted strings. For more information on string substitutions, see Concatination later in this chapter.

GEN

`GEN` *n*`,`*string* `[;`*comments*`]`

The GEN statement passes an instruction string 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 11,EDD.00,TX:15,TO:32000
drivr NOP                        ;program begins here.
        :
        :
      END drivr
```

Note If the length expression *n* is not a simple ATV or number, then string substitution is done on the whole LOD statement including the comments. In general, errors are ignored; however, MACRO insists on matched quotes and will remove unmatched quotes from quoted strings. For more information on string substitutions, see Concatination later in this chapter.

BREAK

`BREAK [;`*comments*`]`

The BREAK command is for use in CDS programming only. It indicates those parts of a CDS program at which natural breaks occur. The loader uses BREAK to construct current page links for off page references. This instruction is required at least every 511 words of code in CDS programs only. See Appendix M for guidelines and examples.

Program Linkage

The linking pseudo operations 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 opcodes are discussed in this section:

ENT
EXT
SEXT
WEXT
ALLOC
RPL

ENT, EXT, SEXT, and WEXT

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

EXT name[='alias'][,name[='alias']...[;comments]]

SEXT name[='alias'][,name[='alias']...[;comments]]

WEXT name[='alias'][,name[='alias']...[;comments]]
```

The ENT pseudo opcode 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 EXT pseudo opcode 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, or EQU or DEF pseudo opcodes. An external symbol can be used with a + (plus) or − (minus) offset or specified as indirect.

The SEXT (soft external) pseudo opcode is useful in defining call macros when the same macro can be used to call an internal or external subroutine. This instruction tells MACRO that, if the symbol is not defined in the module, the symbol is external.

The WEXT (weak external) pseudo opcode sets the "w" (weak external) flag in the external records. Some linkers recognize this flag as an indication that the defining entry point is to be processed, if encountered, but not searched for (in a library) and not reported as an undefined if not satisfactory.

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.

EXT and ENT 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
```

SEXT Example:

```
           CALL  FOO,A,B,C
            :
           SEXT FOO
           JSB FOO
           DEF *+4
           DEF A
           DEF B
           DEF C
           END

           ENT FOO
      P1   NOP
      P2   NOP
      P3   NOP
      FOO  NOP
           JSB .ENTR
           DEF P1
            :
            :
            :
```

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, you 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.

ALLOC

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

The ALLOC pseudo opcode 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 J 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.

An ALLOC statement may specify zero words. If RELOC ALLOC to the space is then done, MACRO will keep track of the highest word generated and use that value as the size.

Example:

```
COM  ALLOC COMMON,0
      .
      .
      .
     RELOC ALLOC,COM
COM1 BSS   40

     ORR        > return to program space
      .
      .
      .
     RELOC ALLOC,COM

COM2 REPEAT   40
     OCT   0
     ENDREP

     ORR
```

MACRO will generate an 80-word ALLOCate common named ‘COM’ and will fill the last 40 words with zeroes.

Example:

```
          NAM   MAIN
Q         ALLOC EMA,50000,2 ; Declare Q to be a 50000 word
                            ; array in EMA.
EMA.ADDR DDEF  Q
           :
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.
GSAV      ALLOC SAVE,100    ; Same SAVE space as in MAIN.
          END
```

Note the following difference between ALLOC and RELOC:

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

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

Example:

```
          NAM   MAIN
          RELOC EMA         ; Use EMA space.
QEMA      BSS   20000       ; Reserve 20000 locally accessible
           :                ; EMA locations.
          END
;
          NAM   SUBR
          RELOC EMA         ; QEMA in SUBR is a different EMA
QEMA      BSS  20000        ; space than QEMA in MAIN.
           :
          END
```

A common use of ALLOC is in a construct like:

```
Q      ALLOC COMMON,100
```

which is equivalent to FORTRAN:

```
COMMON /Q/ARRAY(100)
INTEGER ARRAY
```

RPL

label RPL *instruction_word* [,*value*][;*comments*]

The RPL pseudo opcode 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
.VSUB RPL  101460B,40B       ; This is a two–word replacement.
```

Wherever the loader or generator encounters a JSB .FAD, it is replaced by the octal representation of the machine instruction (105000B). There are some constraints:

- The instruction to be replaced must be a memory reference instruction (that is, 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.

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 opcodes regulate the assembly listing output.

The following pseudo opcodes are discussed in this section:

COL
HED
LIST
SKP
SPC
SUBHEAD
SUP
UNS

COL

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

The COL pseudo opcode 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.

HED

`HED` *heading*

The HED pseudo opcode 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 opcode 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.

Note

The five instructions SKP, HED, SUBHEAD, NAM, or after an END all cause the next line output to the list file to be preceded by:

1. A form feed: one in column one,
2. A page head, and
3. The currently active HED and SUBHEAD lines (if any).

The SPC instruction may also force this condition.

The NAM statement is always listed but SKP, HED, and SUBHEAD statements are only listed if they are in error.

This means that HED and SUBHEAD lines may be put above the NAM by entering them prior to NAM in the source file.

For example:

```
  :                                   :
  :                                 END
END                                 HED ...
                                    SUBHEAD ...
NAM ...                             NAM ...
HED ...                               :
SUBHEAD ...                           :
  :
```

will put the NAM statement on a page by itself with no HED or SUBHEAD.

will put the NAM under the HED and SUBHEAD and continue the program on the same page.

Because no action is actually taken until a line is sent to the LIST file, the order of instructions is irrelevant. For example, in the absence of listing lines, the order of instructions in the following three groups is irrelevant:

```
HED                                           SKP
SUBHEAD           SUBHEAD                     HED
SKP               HED                         SUBHEAD
```

Comments after an END and before a following NAM statement will appear with the module containing the END statement.

Example:

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

Causes the second page of the listing to appear:

```
PAGE#2       S2317.MAC::MANUAL          4:46  PM  TUE,  8 DEC 1987
Support Subroutine #2317

00004  00000 00000  s2317   nop
00005                       end
```

SUBHEAD

SUBHEAD *subheading*

The SUBHEAD pseudo opcode 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 a 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 (except for macro library runs). This table will be in ASCII collating sequence order. Case is not shifted for this sort.

Example:

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

Causes the second page of the listing to appear:

```
PAGE#2     S2317.MAC::MANUAL          4:46  PM  TUE,  8 DEC 1987
Support Subroutine #2317
integer to real conversion

00005  00000 00000  s2317   nop
00006                       end
```

At the end of the listing, the following would appear:

```
Page #  Subhead

2         integer to real conversion
```

Note

The five instructions SKP, HED, SUBHEAD, NAM, or after an END all cause the next line output to the list file to be preceded by:

1. A form feed: one in column one,
2. A page head, and
3. The currently active HED and SUBHEAD lines (if any).

The SPC instruction may also force this condition.

The NAM statement is always listed but SKP, HED, and SUBHEAD statements are only listed if they are in error.

This means that HED and SUBHEAD lines may be put above the NAM by entering them prior to NAM in the source file.

For example:

```
       :                           :
       :                          END
      END                         HED ...
                                  SUBHEAD ...
      NAM ...                     NAM ...
      HED ...                      :
      SUBHEAD ...                  :
       :
```

will put the NAM statement on a page by itself with no HED or SUBHEAD.

will put the NAM under the HED and SUBHEAD and continue the program on the same page.

Because no action is actually taken until a line is sent to the LIST file, the order of instructions is irrelevant. For example, in the absence of listing lines, the order of instructions in the following three groups is irrelevant:

```
HED                                          SKP
SUBHEAD             SUBHEAD                  HED
SKP                 HED                      SUBHEAD
```

Comments after an END and before a following NAM statement will appear with the module containing the END statement.

LIST

LIST *keyword* [;*comments*]

The LIST pseudo opcode 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:

BACK This operand instructs Macro/1000 to restore the prior listing state in effect when the last non-BACK list command, which changed the list state, was executed.

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.

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.

OFF This operand suppresses the assembly listing, beginning with the ‘LIST OFF’ pseudo opcode 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.

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’.

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.

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
```

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

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

PAGE# 1      PROGA.MAC::MANUAL       2:54 PM  THU., 26  MAR., 1981

 00001                MACRO,L,R
 00002                      ; Listing mode is defaulted to short,
 00003                      ; do not 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
```

SKP

```
SKP [;comments]
```

The SKP pseudo opcode 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.

Note

The five instructions SKP, HED, SUBHEAD, NAM, or after an END all cause the next line output to the list file to be preceded by:

1. A form feed: one in column one,
2. A page head, and
3. The currently active HED and SUBHEAD lines (if any).

The SPC instruction may also force this condition.

The NAM statement is always listed but SKP, HED, and SUBHEAD statements are only listed if they are in error.

This means that HED and SUBHEAD lines may be put above the NAM by entering them prior to NAM in the source file.

For example:

```
  :                                       :
  :                                     END
END                                     HED ...
                                        SUBHEAD ...
NAM ...                                 NAM ...
HED ...                                   :
SUBHEAD ...                               :
  :
```

will put the NAM statement on a page by itself with no HED or SUBHEAD.

will put the NAM under the HED and SUBHEAD and continue the program on the same page.

Because no action is actually taken until a line is sent to the LIST file, the order of instructions is irrelevant. For example, in the absence of listing lines, the order of instructions in the following three groups is irrelevant:

```
HED                                                SKP
SUBHEAD             SUBHEAD                        HED
SKP                 HED                            SUBHEAD
```

Comments after an END and before a following NAM statement will appear with the module containing the END statement.

Example:

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

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

SPC

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

The SPC pseudo opcode 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 as if a SKP instruction had been seen. See the previous note for comments related to interaction with other instructions.

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.
```

SUP

```
SUP [;comments]
```

The SUP pseudo opcode 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 opcode is encountered.

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

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

The SUP pseudo opcode 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.

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:

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

UNS

UNS [;*comments*]

The UNS pseudo opcode causes Macro/1000 to resume the printing of lines suppressed by a SUP instruction.

Storage Allocation

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

The pseudo opcodes covered in this section are:

BSS
MSEG

BSS

label `BSS` *#words* `[;`*comments*`]`

The BSS pseudo opcode 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 or ALLOC EMA, 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 a 100-element array.
pointer   DEF ARRAY     ; Pointer into array.
            :
           LDA ARRAY+3   ; Load the fourth element of the array.
```

MSEG

`MSEG` *size* `[;`*comments*`]`

The MSEG pseudo opcode 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 1 to 31 representing the number of pages of the MSEG. It can be an EQU value. Macro/1000 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 operations 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 opcodes are discussed in this section:

ASC
BYT
DEC
DEX
DEY
LIT
LITF
OCT

ASC

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

The ASC pseudo opcode 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 2*n* characters is treated as a comment. If the end of the statement is reached before 2*n* characters have been read, the remaining characters are assumed to be blanks and are stored as such. This statement cannot be continued.

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

Note If the length expression *n* is not a simple ATV or number, then string substitution is done on the whole ASC statement including the comments. In general, errors are ignored; however, MACRO insists on matched quotes and will remove quotes from quoted strings. For more information on string substitutions, see Concatination later in this chapter.

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 opcode 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 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).

Example: Define an array of octal byte constants.

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

DEC

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

The DEC pseudo opcode 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:

1.469368X10E−39 to 1.701412X10E38

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 opcode 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 opcode enters one or more double-precision decimal constants into consecutive words of the object program. It is similar to the DEX pseudo opcode 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 is placed by Macro/1000. If you do not use this command, MACRO stores 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 are stored after it.

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

Example:

```
      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.
```

Example:

```
EMA2  ALLOC  EMA,60000
       :                  ; get double-integer constant
      DLD    D400
      JSB    .DAD
      DEF    VAR          ; add VAR address to it
       :
D400  DDEF   400          ; double-integer constant
VAR   DDEF   EMA2+4400
```

LITF

LITF [;*comments*]

The LITF pseudo opcode is similar to the LIT pseudo opcode (except that MACRO forgets all LITs defined to this point). LITF 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.

When the 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 (unless they were defined by an intervening LIT). Then MACRO forgets about all prior definitions. As a result, subsequent literals will be assigned locations later in the program, even if used prior to LITF.

Example:

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

OCT

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

The OCT pseudo opcode 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 opcodes covered in this section are:

DEF
DDEF
ABS
EQU
DBL
DBR

DEF

label `DEF` *operand* `[;`*comments*`]`

The DEF pseudo opcode 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. (See the subsection of Chapter 2 titled "Legal Uses of Expressions".)
- 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 opcode 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.

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 opcode 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* `[;`*comments*`]`

The DDEF pseudo opcode generates a 32-bit (double-word) relocatable address. The first word is the low-order bits of a 32-bit EMA relocatable address, the second word is the high-order bits. Note that this is the correct order for double-integer math and FORTRAN routines, but is the incorrect order for the Extended Arithmetic Group instructions. (Refer to Chapter 3 of this manual for details of the EAG instructions.)

The label can be used as an operand in a memory reference instruction. It will represent a 16-bit address at which a 32-bit address resides.

The operand field can be:

- A relocatable or an absolute expression in a relocatable program. (See the subsection of Chapter 2 titled “Legal Uses of Expressions”.)
- An operand of the EXT pseudo opcode.
- 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, the result is a word of zeros followed by a 16-bit address.

ABS

$$\texttt{[}\textit{label}\texttt{] ABS} \left\{ \begin{matrix} \textit{integer} \\ \textit{absolute expression} \end{matrix} \right\} \texttt{[;}\textit{comments}\texttt{]}$$

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

The label is optional. If specified, it 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.
```

EQU

label EQU *operand* [;*comments*]

The EQU pseudo opcode 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 aree 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.

For example, if you wish TABLE.A and TABLE.B to occupy contiguous memory locations, you 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
```

Example:

```
Y EQU *
X NOP
```

Now X and Y are equivalenced; that is, 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* [;*comments*]
label DBR *operand* [;*comments*]

The DBR and DBL pseudo opcodes 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. (See the subsection of Chapter 2 titled “Legal Uses of Expressions”.)
- 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.

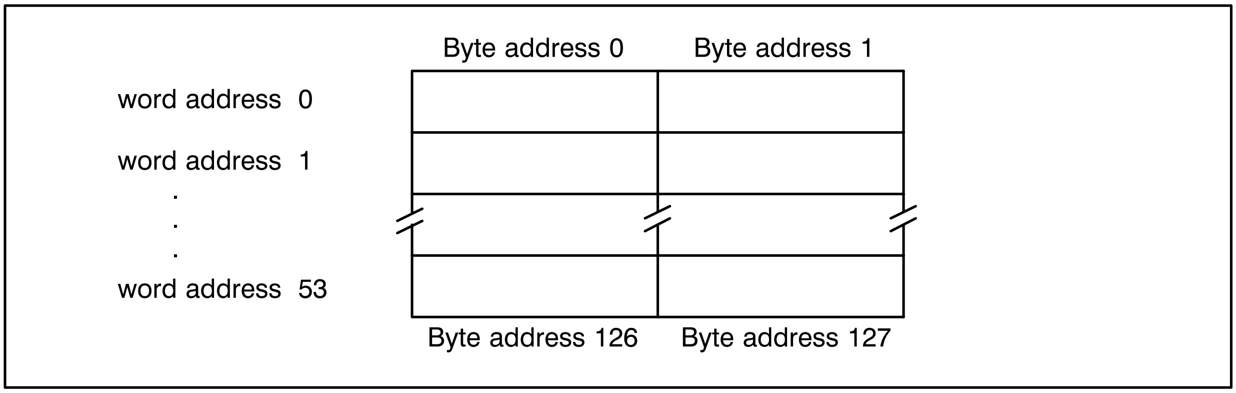


Figure 4-2. Byte Addressing

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 opcode 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.

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: Convert a byte address to a word address.

```
             LDA byte.address     ; Load the byte address.

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

LOADREC

```
    LOADREC   exp1[,exp2   ...]
```

The LOADREC opcode is designed primarily to support preprocessors such as PASCAL which will use it to pass debug and other information. This opcode allows you to generate arbitrary relocatable records. These records may be used as input to a debugger or as code generation records. To be useful, the resulting record must conform to what the loader expects. Normally, this means one of the record structures as found in Appendix H.

The following rules apply:

exp1 Must result in an absolute value. Its double word result is used to bump the program counter for the current Reloc space. The value does not appear in the record itself. This number may be 0.

exp2 Not used. Goes into word 1 (word count) of the record. (See explanation below.)

exp3 Becomes word 2 (checksum) of the record. (See explanation below.)

exp4 Not used. Goes into word 3 of the record.

exp5 Becomes word 4 of the record.

exp6 ... expn Become words 5 through *n*−1 of the record.

When the record is completed, word 1 is changed to the correct record word count (see Appendix H). The record checksum is then computed and put into word 2 of the record as required by the standard record format.

The maximum size of a record is 127 words so only the first 128 expressions will be processed.

The expressions may result in other than absolute values. Only the low order 16 bits of the expression value, however, will be put into the record. If the high order part is desired, the expression should be logically shifted right by 16 bits (:lsh:−16).

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 opcodes 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 “&.” characters. Some system ATVs are available for you to use. These are discussed in Appendix K.

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 pseduo opcode.
- In the comments field, except that of the NAM, LOD, GEN, and ASC 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 opcodes 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        [;comments]
label [num,size]   IGLOBAL    assembly-time variable    [;comments]

label [num,size]   CLOCAL     character expression      [;comments]
label [num,size]   CGLOBAL    assembly-time variable    [;comments]
```

where:

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.

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 value of &var declared in OUTER.
      ENDMAC
```

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* [;*comments*]

label CSET *character expression* [;*comments*]

The ISET and CSET pseudo opcodes 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 opcode.

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

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	−, :ICH:, :L:, :MR:, :NOT:, :S:, :SY:, :T:, and :UC:
arithmetic operators	*, /, +, −, :ASH:, :LSH:, :MOD:, :ROT:
comparison operators	=, >=, <=, <>, > and <
logical operator	:AND:, :OR:

Concatenation is also an operation in Macro/1000.

The operators :ICH:, :L:, :S:, :T:, and :UC: are macro pass operators. They are expanded at the same time that ATVs are expanded. This expansion occurs before any other operation or expression evaluation.

The remaining operators are unique to:

- The macro expansion pass or
- The assembly pass or
- Are available in both passes.

However, the operators are different in that macro pass numeric expressions are evaluated to 16 bits while assembly pass expressions are evaluated to 32 bits. Macro pass expressions are those that appear in the following statements:

IGLOBAL
ILOCAL
CGLOBAL
CLOCAL
REPEAT
ISET
CSET
AIF
AELSEIF
AWHILE

All other expressions are assembly pass expressions.

Assembly-Time Expressions

Assembly time expressions are evaluated to 32 bits. Each expression results in a value (32-bit signed integer) and zero or more sets of relocation space attributes. There are six relocation spaces:

program
base page (in CDS this is the LOCAL space)
common
pure code (CDS only)
EMA
save

In addition there are any number of external spaces. These are referenced by the following instructions:

ALLOC EMA
ALLOC SAVE
ALLOC COMMON
EXT

The external spaces all have symbol id numbers assigned to them when they are first encountered by MACRO.

The result of an expression may have at most one external symbol id number in its relocation space attribute list. In addition, it may have any number of the six relocation spaces in its relocation space attribute. A count is kept of the relocatability in each relocation space. This count increases if a member from that space is added to the expression and decreases if a member is subtracted from that space.

For example, if:

```
      reloc program
prog  equ *+10
      reloc base
base  equ *
      reloc common
com   equ *
      reloc ema
ema   equ *
      reloc save
save  equ *
      ext ext
```

and COMBO is defined as:

```
COMBO equ prog+base+com+ema+save+ext
```

COMBO has relocatability of +1 in each of the following spaces:

program
base page
common
EMA
save
external

As another example:

```
    prog2  equ  prog+prog
```

has relocatability of +2 in the program space (and zero or none in all others).

```
          reloc program
   prog1 equ prog2-*
```

has relocatability of +1 in program space (the −* decreased it by 1).

Note that in all cases, the expression also resulted in a value that was the indicated sum or differences of the values of the terms involved.

The result of any expression can always be saved in an equate (EQU) symbol as per COMBO above. Other than this, there are restrictions on the relocatability of expressions depending on the opcode with which they appear. These restrictions are as follows:

1. All expressions and partial expressions may have relocatability in at most 1 external space. They may, however, have multiple relocatability in that space, therefore:

   ```
          ext bar
   foo    alloc common,10
   *
   good   equ foo+foo    ; Double relocatability in external space
                         ; foo, OK.
   bad    equ foo+bar    ; Bad, only one external space allowed in an
                         ; expression
   ```

2. Defs and implied defs (from =L literals) must have expressions that exist in at most one relocatable space, however, the relocatability in that space can be in the range −8 to +7, inclusive.

3. DBL and DBR expressions may be in at most one relocation space and must have relocatability in that space of −1 to +1, inclusive.

4. All other opcodes restrict their expressions to have either 0 or 1 relocatability in at most one space. (Some, such as ABS, are more restrictive.)

Examples:

```
         ext bar
   foo   asc 10,A string of char.
          :
          :
         def foo+foo   ; byte address of string foo.
         dbl foo       ; same as above.
         def foo+foo+1 ; byte address of space after the A in
                       ; string foo.
          :
          :
         ldb =l(foo+foo)  ; byte address of foo to B-Register.
          :
          :
         adb =l(–foo–foo) ; subtract byte address of foo.
          :
          :
         ldb =l(bar+bar)  ; byte address of external bar.
          :
          :
         ada =l(–bar)     ; subtract address of bar
```

In summary, assembly time expressions may refer to, at most, one external. In addition, they may refer to multiple non-external relocation spaces a multiple number of times. The result of any such expression may be saved by means of an EQU instruction, however, code production limits the number of relocation spaces to, at most, one for all code producing instructions. Further, with the exception of DEFs and implied defs (from the =L literal), multiple relocation in a space is not allowed. For DEFs and implied DEFs, the multiple is limited to −8 to +7, inclusive.

As a practical matter, the only common use for multiple relocatability is in dealing with byte addresses and in doing assembly time negation of addresses.

The Operators

The following operators are available only in macro pass expressions:

=
>=
<=
<>
>
<

The following operators are available only in assembly time expressions:

:MR:
:SY:

The following operators are available either in macro or assembly passes:

+
−
*
/
:AND:
:ASH:
:LSH:
:MOD:
:NOT:
:OR:
:ROT:

Parentheses ( ) may be used in any expression. Expressions are evaluated left-to-right with parentheses dictating the only precedence.

Unary Operators

The unary operators are:

–	(negate)
:ICH:	(integer equivalent of character)
:L:	(length)
:MR:	(memory relocatability)
:NOT:	(negate)
:S:	(substring)
:SY:	(symbol ID)
:T:	(type)
:UC:	(upper case)

Negate (–)

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

Examples:

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

Integer Equivalent of a Character (:ICH:)

:ICH: takes one or two characters and converts them into their integer equivalents as follows:

```
LDA = L(–:ICH:“A”)          ;get negative of “A”
```

For a more detailed use of :ICH:, refer to the SQUZ macro code in Chapter 5.

Length Operator (:L:)

The length operator 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 operator, 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.
```

Memory Relocatability (:MR:)

The unary operator :MR: returns the relocatability of its operand as an absolute value, as follows:

0	= Absolute
1	= Program relocatable
2	= Base page relocatable
3	= Common relocatable
4	= Pure code relocatable (CDS only)
5	= EMA relocatable
6	= SAVE relocatable
7	= External
9	= Allocate EMA
10	= Allocate SAVE
12	= Allocate common
20	= Two or more of the above

Logical Negation (: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.
        LDA =L (:NOT:10)         ; gets –11
```

Substring Operator (:S:)

The substring operator is replaced at assembly time by a portion of a type-character assembly-time variable, macro parameter, or string. The operator 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.

Note The substring operator will consume strings that follow it up to another operator or to the end of the line, as follows:

```
:s:[1,2]’WHAT’& WHO’WHEN’
```

results in: `“WH”`

while: `:s:[1,2]“WHAT”:uc:&WHO“WHEN”`

where: `&WHO is “HIM”`

results in: `“WHHIMWHEN”`

Example:

```
&substring[0,19] CGLOBAL “XXXXXXXXX”        ; declare variable
                                            ; to hold substring.
&string           CGLOBAL ’Macro Assembler’ ; declare string
&substring        CSET    :S:[7,9]&string   ; substring is now
                                            ; “Assembler”.
```

Symbol ID (:SY:)

The unary operator :SY: returns, as an absolute value, the symbol ID of the expression it operates on.

This is the same number as in the external and allocate records for the symbol.

If the expression does not reference an external or allocate symbol, 0 is returned.

Type Operator (:T:)

The type operator is replaced by the current type of the assembly-time variable that follows it. A character string always results from use of the type operator. 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
```

Uppercase Operator (:UC:)

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

Example:

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

Arithmetic Operators

There are eight arithmetic operators:

*
/
+
−
:ASH:
:LSH:
:MOD:
:ROT:

When appearing in the operand field as operators, the arithmetic operators *, /, +, and − perform multiplication, division, addition, and subtraction on numerical values. When used in assembly time expressions, the + and − operators accept any operands. All other operators in this group require both operands to be absolute.

:ASH: and :LSH:

The :ASH: operator performs an arithmetic shift and the :LSH: operator performs a logical 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. In macro pass expressions, these are 16-bit operators, while in Assembly pass expressions, they are 32-bit operators.

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.
foo    EQU BAR:LSH:&IT
```

: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 5 (value of 2).
```

: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. In macro pass expressions, 16 bits are rotated, while in Assembly pass expressions, 32 bits are rotated.

Example:

```
&ROT IGLOBAL 7         ; &ROT is declared and rotated right
&ROT ISET &ROT:ROT:-2  ; two bits.
     ABS foo:ROT:3
```

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 are interpreted as strings, also. Similarly, if the first operand is an integer, the Assembler interprets 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
```

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. In macro pass expressions, these are 16-bit operators. In Assembly pass expressions, they are 32-bit operators, and both operands must be absolute.

Examples:

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

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”.

Example:

```
&word1    CLOCAL  ’one and’
&word2    CLOCAL  ’ two’
&word12   CLOCAL  &word1&word2 ; &word12 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 operations 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 opcode 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 opcodes is:

opcode operand [;*comments*]

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

opcode [;*comments*]

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:	(uppercase)
	:ICH:	(character to integer)		
• 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)

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:	:ICH:

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 opcodes must evaluate to an integer. 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 ORed 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')
```

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. This nesting limit is global, that is, if the level is at 14 and a macro call is made to a macro that does two more AIFs, an error results. Keep this in mind when coding recursive macros.

Example 1:

```
AIF  &debugflag = 'ON'  ; If flag is on, then assemble the call
   WRITE temp1,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                       ; If &status is not 1, 2, or 3
   ERROR &status            ; assemble the call to macro ERROR.
AENDIF
```

Using AWHILE

MACRO continues 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 eventually evaluates to 0 or the assembler will be in an infinite loop.

AWHILE statements can be nested to a level of five deep. This nesting limit is global, that is, if the level is at 3 and a macro call is made to a macro that does two more AWHILEs, an error results. Keep this in mind when coding recursive macros.

Examples:

```
asciitab  EQU *
&X        IGLOBAL 64
          AWHILE &X < 90
&X         ISET &X+1
           DEC &X
          AENDWHILE
```

is the same as typing:

```
asciitab   DEC 65
           DEC 66
           DEC 67
            :
           DEC 90
```

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. The original &.REP value is 1, and it is incremented by 1 each time the loop is repeated.

REPEAT and ENDREP

```
REPEAT   #repetitions   [;comments]

ENDREP [;comments]
```

The REPEAT pseudo opcode 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. The original &.REP value is 1, and it is incremented by 1 each time the loop is repeated.

MNOTE

`MNOTE` *string_expression*

The MNOTE pseudo opcode 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. *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 (assumes &REG=Q):

```
    MNOTE Register should be A or B, not Q
      ^
60>> User-defined error
```

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, including macro statement formats, special considerations of comments and listing options within the definition, and redefinition of opcodes.
- Macro parameters, including how to pass and receive information to and from macro calls, formal, actual, and default parameters.
- Nesting macros.
- Creating macro libraries, including 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, are 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 optional parameter list can contain one 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. Their usage is determined by the macro definition.

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

From the previous example, 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 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 MACRO to search that library if a macro which has not been defined in the program is encountered.

The format of the MACLIB statement is:

[*label*] MACLIB *filedescriptor*

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

The operand field contains an RTE file name. The file name 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.

In the CI file-system environment, MACRO will default to user-defined search path 3 (refer to the UDSP description in the CI User's Manual for details on UDSPs). If this fails, MACRO then searches the directory that contains the source file.

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 with the same name.

Libraries are 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:

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

If a label 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 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.

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
```

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 K.

The previous example is another method of ensuring that a unique label will be generated in each expansion. It defines 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: 6/17/92  Date changed: 7/3/92  *
.* 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 the 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 ENDMAC statement 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 macro parameters 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           &16B               &32767
    &VariABle           &X25f1             &label
```

Note that lowercase characters are mapped to uppercase 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

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.

If you were to have a label in the macro body and you were to call a 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 such as:

```
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, 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.

If a character string contains a single quote, the string must be surrounded by single quotes and two single quotes are required in the string for one to be passed as part of the string. Therefore, to produce a single quote by itself, four single quotes are required.

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 value of the location counter when the macro call is made. If the macro does not change the location space with an ORB, ORR, ORG, or RELOC statement, this will be the first word of the expanded macro.

Examples of 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`
`STOP!` All special characters are legal; parameters containing any special characters listed above must be surrounded by single quotes.

`&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.

`'Don''t Care'` pass the string Don’t Care. Notice that it takes two quotes for one to be passed as part of the string.

`' ' ' '` character string of one single quote. Notice that four single quotes produce one quote.

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.

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.

Example:

```
    MACRO
    &label    MOVE    &from,&to,&REG’A’
    &label    LD&REG  &from
              ST&REG  &to
    ENDMAC
```

All of these macro call statements 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.

Redefinition of Opcodes

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

```
            MACRO
            MPY &PAR1
            JSB MULTI
            DEF MULTIRTN&.Q
            DEF &PAR1
MULTIRTN&.Q 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 is expanded as a macro, and the remaining opcodes are 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 following example illustrates the use of the :ICH: operator as well as other constructs.

The following macro is used to generate squoze code for an instruction symbol table. Squoze code is radix 40 (50b), where we assign the value 0 to blank, 1-10 to 0-9, 11-36 to A-Z, 37,38,39 to ., $, and _.

This macro generates a one or two word (two if three or more characters) squoze code symbol, followed by the opcode. The symbol table scanner knows there is a second word if the third character in the first word is not blank.

```
      MACRO
      INST &SYM,&OP
*
*
*                    !  ″  #  $  %  &  ′  (  )  *  +  ,  -
&SQTBL[64] IGLOBAL 0,[3]-1,   38,[9]-1,                     \
                  37,-1, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,      \
                  [7]-1,         11,12,13,14,15,16,17,18,19,\
                  20,21,22,23,24,25,26,27,28,29,30,31,32,33,\
                  34,35,36,[4]-1,39
*
*
&N    ILOCAL :L:&SYM
&C[6] ILOCAL  [6]0
&X[6] ILOCAL  [6]0
&J    ILOCAL  0
&SYN  CLOCAL  &SYM
*
      AWHILE &N>0
&J      ISET :ICH::S:[&N,1]:UC:&SYN-37B  ; integer value of char less 37B
*                                        ; entry 1 in table is for blank
&C[&N]  ISET &SQTBL[&J]                  ; get squoze code for char.
&X[&N]  ISET 1                           ; Note occurrence
&N      ISET &N-1
      AENDWHILE
*
* Want c1 for one char., c1*40+c2 for two, and (c1*40+c2)*40+c3 for three.
* Note, expression analyzer is strictly left to right, not precedence.
*
        ABS &C[1]*(&X[2]*39+1)+&C[2]*(&X[3]*39+1)+&C[3]
        AIF :l:&sym > 2    ; Need second word only if more than 2 char.
          ABS &C[4]*(&X[5]*39+1)+&C[5]*(&X[6]*39+1)+&C[6]
        AENDIF
        OCT &OP           ; include the opcode
     ENDMAC
```

Recursion

A macro can invoke itself by being called from within its own definition.

Example:

```
         MACRO,l=l
                  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.

Also note that the nesting level of AIFs (16) and AWHILEs (5) must not be exceeded. In the above macro, a call with 17 would exceed the AIF nest limit of 16.

```
PAGE# 1       FACT:MAC::MANUALS      8:51 AM  TUE.,  1  SEPT, 1987
00001                MACRO,l=l
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:  Macro/1000  Rev.  5000  870429 : No  errors found
```

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. By referencing the library in a source file with a MACLIB statement, you can access all the macros in that library.

Note that a maximum of five macro libraries is allowed per program.

The ‘M’ option in 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 runstring specifies the name of a macro library and not the relocatable file name.

The ‘T’ option in the control statement takes on a new meaning when used with the ‘M’ option. ‘T’ normally means to list the symbol table, when used with ‘M’, it causes a list of all the macro names in the library to be placed in the list file.

In the Macro Library creation mode, the number of opcodes available (outside of the macros) is limited to the following:

INCLUDE
MACLIB
DELETE
EXTRACT
SUBHEAD
HED
SKP
SPC

Except for DELETE and EXTRACT (described below), all of these opcodes are similar to those of the same name discussed elsewhere in this manual, however, the following differences do exist:

1. No ATVs are allowed, therefore, there are no string substitutions done.
2. Only one level of INCLUDE is allowed, however, the INCLUDE file may have one or more MACLIB request.

DELETE and EXTRACT

DELETE *name*[,*name*][,*name*]...[*name*][;*comments*]

EXTRACT *name*[,*name*][,*name*]...[*name*][;*comments*]

EXTRACT or DELETE statements must immediately follow an INCLUDE or MACLIB statement. While as many EXTRACT or DELETE statements as needed may be used, they must all be either EXTRACTs or DELETEs. That is, both EXTRACT and DELETE may not appear after the same INCLUDE or MACLIB statement.

The macros named in EXTRACT statements are (if found) included in the library being built, while all other macros in the INCLUDE or MACLIB file are excluded.

The macros named in DELETE statements are excluded from the library being built, but all the other macros in the INCLUDE or MACLIB file are included.

The domain of the EXTRACT and DELETE statements is the INCLUDE or MACLIB file they appear in. For example, an INCLUDE y file that refers to a MACLIB containing the macro 'CALL' may extract call as follows:

```
INCLUDE y
EXTRACT CALL
```

where y might, for example, be $MACLB or $CDSLB.

Note Some caution is required so that macros called by the extracted macro are also extracted.

The EXTRACT and DELETE statements make it easy to edit a macro library even if you don't have a copy of source available to you.

Procedure to Create a Macro Library

The following is a procedure to follow to create a macro library. Some sample macros are provided.

First, 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 places the macros in the file $LIB. Any of the macros in that file can be accessed by another program via the MACLIB $LIB statement.

Example:

Macros obtained from an INCLUDE or MACLIB file, are listed to the list file (if listing is on). This is a way to get a listing from a MACLIB file, even without the source listings. The following:

```
MACRO, M, L
    MACLIB  $MACLIB
    END
```

will produce a listing of $MACLIB.

A

Assembler Error Messages

Introduction

This appendix contains assembler error messages that Macro/1000 could issue as it assembles a source file. When Macro/1000 detects an error, it prints the error number and description on the list device or file. At the end of compilation, Macro/1000 prints an error summary including the line numbers where errors occurred and the total number of errors encountered.

Error conditions are returned through the P-type FMGR global 1P. See the Terminal User's Manual for more information on P-type globals. In CI, this variable is referred to as RETURN1.

If there are any errors, and Macro/1000 was asked to produce a binary file, it will purge that file.

Error Numbers/Descriptions

The following is a description of the Macro/1000 error messages in numerical order.

Error # **Description**

```
 1   >> Illegal file namr
 2   >> Include files may not be nested past five (one in maclib build) deep
 3   >> Undefined Macro error code.  Try RU,MACRO,–1
 4   >> Opcode illegal in absolute assembly
 5   >> Greater than 1/4 million symbols used. Can’t give symbol table dump
 6   >> MACLIB file must be type 1.
 7   >> Symbol table paging file overflow.
 8   >> Intermediate data file overflow.
 11  >> Corrupt macro library file.
 12  >> Include file can’t be type 1,2, or 5.
 14  >> Illegal listing file type.
 21  >> Old macro library.  Try: ’MACRO,–3,,<maclib>’
 47  >> SEXT external may not be used in a pass two expression
 48  >> MVW, MBT, CMW, and CBT are illegal opcodes in CDS code space.
```

Error # **Description**

```
 49  >> Matching quote not found.
 50  >> Commas are allowed only in ASG and SRG instruction opcodes.
 51  >> This pseudo op must appear before any code or data is assembled.
 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
 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 may not 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 define two word RPL values
 64  >> End of file found before AENDWHILE or ENDREP
 100 >> &.PRAM[n] index missing or out of range.
 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 integer > 0
 105 >> Length of string > 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
 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, default to 0
 118 >> ATV array size must be <= 80 and > 0
 119 >> Array subscript must be surrounded by square brackets
 120 >> Array subscript may 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 cannot be <= 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 is ignored, default to one word
 127 >> Too many elements in ATV array declaration; excess ignored
 128 >> No operand in CGLOBAL/CLOCAL, default to null string
 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
```

Error #   Description

```
156 >> String must be <= 80 character, 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 may not 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 >> Can not use both EXTRACT and DELETE following INCLUDE or MACLIB
168 >> Only five macro libraries allowed per program
201 >> Opcode missing
202 >> Line too long after string substitution
203 >> Column indicators should be three integers separated by commas
204 >> Mnemonic field longer than 16 characters
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 cannot 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
227 >> Number in substring must be >= 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 may not 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 <= 125 words
236 >> Legal string compare operators are = and <>
237 >> Line continuation may not start before the operand field
```

Error # **Description**

```
 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
 246 >> String in expression must be less than 5 characters long
 247 >> BASE page or LOCAL address is not in range 0 to 1023
 250 >> Double integer overflow
 251 >> Illegal column indicator in COL statement
 252 >> Keyword must be ON, OFF, BACK, SHORT, MEDIUM, or LONG
 253 >> Octal integers can not contain an 8 or 9
 254 >> Literals not legal on this opcode
 255 >> Expected ALLOcate,BASE,CODE,COMMon,DATA,LOCAL,PROGram,SAVE, or STATic
 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 =A, =B, =D, =F, =J, =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 >= 1 and <= 31
 278 >> Syntax error in ALLOC
 279 >> EMA and ALLOC EMA or MSEG cannot 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 may only be used 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
```

Error #	Description
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
294 >>	Symbol after RELOC ALLOC must be an ALLOC symbol.
295 >>	Relocation space just closed is too small for the code generated.
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 may not reference the same symbol
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 =F, =J or =S(3+ chars.) literal allowed on this opcode
321 >>	Only =A,=B,=D,=L,=R, or =S(1-2 chars) literal allowed on this opcode
322 >>	NAM comment may not exceed 73 characters in length
323 >>	EQUs may not be negative when ’ASMB’ is the control statement
324 >>	BSS,COM,RELOC,ORB,ORG,or machine insts. may not appear before NAM
325 >>	NAM or ORG statement missing
326 >>	=L/=J symbols must be previously defined unless MRG in CDS code space
327 >>	Illegal relocatable record encountered at address
328 >>	Off page reference in an MR instruction found at address
329 >>	Corrupt record found in relocatable file at address

B

Macro/1000 Instruction Set

Introduction

This appendix summarizes the machine instructions and pseudo operations 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
- CDS Code

Pseudo Operations

- 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
- Error Reporting
- CDS Control
- Backward Compatibility
- Miscellaneous Other

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.

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.	

Shift/Rotate instructions can be combined as follows:

A-Register Instructions

$$\left[\left\{\begin{matrix}\texttt{ALS}\\ \texttt{ARS}\\ \texttt{RAL}\\ \texttt{RAR}\\ \texttt{ALR}\\ \texttt{ALF}\\ \texttt{ERA}\\ \texttt{ELA}\\ \texttt{SAE}\\ \texttt{LAE}\end{matrix}\right\}\right]\texttt{[,CLE] [,SLA]}\left[\left\{\begin{matrix}\texttt{,ALS}\\ \texttt{,ARS}\\ \texttt{,RAL}\\ \texttt{,RAR}\\ \texttt{,ALR}\\ \texttt{,ALF}\\ \texttt{,ERA}\\ \texttt{,ELA}\\ \texttt{,SAE}\\ \texttt{,LAE}\end{matrix}\right\}\right]$$

B-Register Instructions

$$\left[\left\{\begin{matrix}\texttt{BLS}\\ \texttt{BRS}\\ \texttt{RBL}\\ \texttt{RBR}\\ \texttt{BLR}\\ \texttt{BLF}\\ \texttt{ERB}\\ \texttt{ELB}\\ \texttt{SBE}\\ \texttt{LBE}\end{matrix}\right\}\right]\texttt{[,CLE] [,SLB]}\left[\left\{\begin{matrix}\texttt{,BLS}\\ \texttt{,BRS}\\ \texttt{,RBL}\\ \texttt{,RBR}\\ \texttt{,BLR}\\ \texttt{,BLF}\\ \texttt{,ERB}\\ \texttt{,ELB}\\ \texttt{,SBE}\\ \texttt{,LBE}\end{matrix}\right\}\right]$$

Register Reference, Alter/Skip Group

Opcode	Instructions
CCA	Clear and complement A.
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, unconditionally skip the next instruction.
SEZ	Skip if E is zero.
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.

(No operands in this group.)

Alter/skip instructions can be combined as follows:

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

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

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 group.)
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.	

Opcode	Instructions	Operand Format
XAX	Exchange X and A.	(No operands in this group.)
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.
JLA	Jump and load A.	[@]m
JLB	Jump and load B.	[@]m
JLY	Jump and load Y.	[@]m
JPY	Jump indexed by Y.	m

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.

Opcode	Instructions	Operand Format
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 I/O control bit.	Select code.
CLCC	Clear I/O control bit and flag bit.	Select code.
CLF	Clear I/O flag bit.	Select code.
SFC	Skip if I/O control bit is zero.	Select code.
SFS	Skip if I/O control bit is one.	Select code.
STC	Set I/O control bit.	Select code.
STCC	Set I/O control bit, clear flag bit.	Select code.
STF	Set I/O flag bit.	Select code.
CLO	Clear Overflow bit.	No operand.
SOC	Skip if Overflow bit is zero.	No operand.
SOCC	Skip if Overflow bit is zero, clear flag.	No operand.
SOS	Skip if Overflow bit is one.	No operand.
SOSC	Skip if Overflow bit is one, clear flag.	No operand.
STO	Set Overflow bit.	No operand.
HLT	Halt the computer.	Select code of flag bit.
HLTC	Halt the computer, clear flag bit.	Select code.

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.
MPYD	Multiply with A, double format.	[@]m or literal.
DIV	Divide with A and B.	[@]m or literal.
DIVD	Divide with A and B, double format.	[@]m or literal.
ASL	Arithmetic shift A and B left.	integer, number of bits to shift.
ASR	Arithmetic shift A and B right.	integer, number of bits to shift.
LSL	Logically shift A and B left.	integer, number of bits to shift.
LSR	Logically shift A and B right.	integer, number of bits to shift.
RRL	Rotate A and B left.	integer, number of bits to rotate.
RRR	Rotate A and B right.	integer, number of bits to rotate.
SWP	Swap A and B.	No operand.

Floating-Point Instructions

Opcode	Instructions	Operand Format
FAD	Floating point add to A and B.	[@]m or floating point literal.
FDV	Floating point divide to A and B.	[@]m or floating point literal.
FIX	Convert floating point to fixed point.	No operand.
FLT	Convert fixed point to floating point.	No operand.
FMP	Floating point multiply to A and B.	[@]m
FSB	Floating point subtract to A and B.	[@]m

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.
MBF	Move bytes from alternate map.	No operand.
MBI	Move bytes into alternate map.	No operand.
MBW	Move bytes within alternate map.	No operand.
MWF	Move words from alternate map.	No operand.
MWI	Move words into alternate map.	No operand.

Opcode	Instructions	Operand Format
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.

CDS Code

PCAL	Procedure call.	Name and call sequence parameters.

Pseudo Operations

Assembler Control

Opcode	Instructions	Operand Format
NAM	Name relocatable program.	Name plus optional parameters.
ORG	Establish program 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.
DELETE	Delete listed macros from included macro file.	Macro name list.
EXTRACT	Include only named macros from included macro file.	Macro name list.

Loader and Generator Control

Opcode	Instructions	Operand Format
LOD	Define loader record.	Number of characters followed by loader record.
GEN	Define generation record.	Number of characters followed by generation record.

Program Linkage

ENT	Define entry point.	name[=‘alias’][,name...]
EXT	Define external routine.	name[=‘alias’][,name...]
WEXT	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	Print heading at top of page.	Heading.
LIST	Specify list option.	Keyword.
SKP	Skip to top of next page.	No operand.
SPC	Skip n lines of listing.	No operand.
SUBHEAD	Specify a subhead at top of page.	Subheading.
SUP	Suppress extended code list.	No operand.
UNS	Resume extended code list.	No operand.

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 address.	m or literal.
DBR	Define right byte address.	m or literal.

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.
ISET	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.

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.

CDS Control

Opcode	Instructions	Operand Format
CDS	Turn on/off CDS mode	Keyword.
BREAK	Generate break record	No operand.
LABEL	Define CDS label	Name of a procedure.

Backward Compatibility

Opcode	Instructions	Operand Format
ORB	Relocate code to base page	No operand.
ORR	Return to previous relocation space	No operand.
IFN	Conditional assembly	No operand.
IFZ	Conditional assembly	No operand.
XIF	Conditional assembly	No operand.
REP	Repeat following line	Count expression.
COM	Blank common	Name (size) list.
EMA	Old EMA declaration	Size, MSEG.
UNL	Turn off listing	No operand.
LST	Turn on listing	No operand.
MIC	Define micro opcode	No operand.
RAM	Define micro opcode	No operand.

Miscellaneous Other

Opcode	Instructions	Operand Format
LOADREC	Generate arbitrary relocatable	Integer expressions.

HP 1000 Computer Instruction Set (Octal Opcode)

Table C-1 is a listing of the instruction mnemonics that are available on the HP 1000 series of computers in “ASCIIbetical” order. That is, mnemonics with dollar sign ($) leading characters are listed first, mnemonics with dot (.) leading characters are listed next, and so forth. Table C-2 is a listing of instruction mnemonics in opcode (octal) order.

No single, definitive guideline exists as to when an instruction mnemonic is preceded with a dot. In many cases, the same mnemonic may be used with or without a dot (denoted with an asterisk in Tables C-1 and C-2). It cannot be assumed, in all cases, that a mnemonic without a dot and the same mnemonic with a dot will relate to the same instruction. As an example, XADD and .XADD represent two entirely different instructions. *In general*, if an instruction is recognized by Macro/1000 or is user-callable from FORTRAN, it is not preceded by a dot. Instructions not recognized by Macro/1000 are *generally* preceded by a dot. In other instances, the only rationale for the presence (or absence) of a leading dot is historical precedence.

In some cases, an instruction may be represented by as many as three mnemonics.

Instruction Mnemonics in ASCIIbetical Order

Table C-1 was taken primarily from the Replace Instruction (RPL) files for both the RTE-A and RTE-6/VM operating systems. A mnemonic may appear in the “All 1000s” column if the opcode of the instruction it represents is the same in all computers (CPUs and SPUs) in which the instruction is implemented. It should be noted, however, that this does not mean that all HP 1000 computers can execute the instruction. As an example, the tangent function (TAN) is not implemented on A400, A600, A600+, or E-Series computers but the opcode of the instruction is 105320 (octal) on all computers in which it is implemented.

Some instructions have been implemented only on later versions of a computer series (for example, the F-Series). Special or custom firmware (for example, double integer and third-party firmware) is not included in Table C-1.

Note The instruction set listed in each computer series reference manual takes precedence over Table C-1 and therefore should be considered as the ultimate source for instruction sets.

Table C-1. Instruction Mnemonics in ASCIIbetical Order (sheet 1 of 9)

Mnemonic	HP 1000 Computer Series					
	All 1000s	All A	A990	E,F	E	F
$LIBR	–	100701	–	–	105340	105340
$LIBX	–	100702	–	–	105341	105341
$LOC	–	–	–	–	105241	105241
$PRIV	–	100711	–	–	–	–
$SETP	105227	–	–	–	–	–
$SJP	–	100703	–	–	–	–
$SJS0	–	100704	–	–	–	–
$SJS1	–	100705	–	–	–	–
$SJS2	–	100706	–	–	–	–
$SJS3	–	100707	–	–	–	–
..DCM	–	–	–	105216	–	–
..FCM	–	105232	–	–	–	105232
..MAP	–	–	–	105222	–	–
..TCM	105233	–	–	–	–	–
.ADQA	–	101413	–	–	–	–
.ADQB	–	105413	–	–	–	–
.ADX *	105746	–	–	–	–	–
.ADY *	105756	–	–	–	–	–
.BLE	–	105207	–	–	–	105207
.CACQ	–	101407	–	–	–	–
.CBCQ	–	105407	–	–	–	–
.CAX *	101741	–	–	–	–	–
.CAY *	101751	–	–	–	–	–
.CBS *	105774	–	–	–	–	–
.CBT *	105766	–	–	–	–	–
.CBX *	105741	–	–	–	–	–
.CBY *	105751	–	–	–	–	–
.CAZ	–	101411	–	–	–	–
.CBZ	–	105411	–	–	–	–
.CCQA	–	101406	–	–	–	–
.CCQB	–	105406	–	–	–	–
.CFER	105231	–	–	–	–	–
.CIQA	–	101412	–	–	–	–
.CIQB	–	105412	–	–	–	–
.CMW *	105776	–	–	–	–	–
.CPM	–	105236	–	–	105352	105352
.CPU	–	105300	–	–	–	–
.CPUID	–	105300	–	–	–	–
.CXA *	101744	–	–	–	–	–

Note: * May be used with or without a leading dot

Table C-1. Instruction Mnemonics in ASCIIbetical Order (sheet 2 of 9)

Mnemonic	HP 1000 Computer Series					
	All 1000s	All A	A990	E,F	E	F
.CXB *	105744	–	–	–	–	–
.CYA *	101754	–	–	–	–	–
.CYB *	105754	–	–	–	–	–
.CZA	–	101410	–	–	–	–
.CZB	–	105410	–	–	–	–
.DAD	–	105014	–	–	105321	105014
.DCO	–	105204	–	–	105324	105204
.DDE	–	105211	–	–	105331	105211
.DDI	–	105074	–	–	105325	105074
.DDIR	–	105134	–	–	105326	105134
.DDS	–	105213	–	–	105333	105213
.DFER	105205	–	–	–	–	–
.DIN	–	105210	–	–	105330	105210
.DINT	–	–	–	–	–	105101
.DIS	–	105212	–	–	105332	105212
.DIV *	100400	–	–	–	–	–
.DIVD *	–	–	104100	–	–	–
.DLD *	104200	–	–	–	–	–
.DMP	–	105054	–	–	105322	105054
.DNG	–	105203	–	–	105323	105203
.DSB	–	105034	–	–	105327	105034
.DSBR	–	105114	–	–	105334	105114
.DSPI	–	–	–	–	105357	105357
.DST *	104400	–	–	–	–	–
.DSX *	105761	–	–	–	–	–
.DSY *	105771	–	–	–	–	–
.DSZ	–	–	105222	–	–	–
.DVCT	–	–	–	–	–	105460-000NN0
.ENTC	–	105235	–	–	105356	105356
.ENTN	–	105234	–	–	105354	105354
.ENTP	105224	–	–	–	–	–
.ENTR	105223	–	–	–	–	–
.ETEQ	–	–	–	–	105353	105353
.EXIT0	–	105417	–	–	–	–
.EXIT1	–	105415	–	–	–	–
.EXIT2	–	105416	–	–	–	–
.FAD *	105000	–	–	–	–	–
.FDV *	105060	–	–	–	–	–
.FIX *	105100	–	–	–	–	–
.FIXD	105104	–	–	–	–	–

Notes: * May be used with or without a leading dot
N Any octal numeric

Table C-1. Instruction Mnemonics in ASCIIbetical Order (sheet 3 of 9)

Mnemonic	HP 1000 Computer Series					
	All 1000s	All A	A990	E,F	E	F
.FLT	105120	–	–	–	–	–
.FLTD	105124	–	–	–	–	–
.FLUN	105226	–	–	–	–	–
.FMP *	105040	–	–	–	–	–
.FNW	–	–	–	–	105345	105345
.FPWR	105334	–	–	–	–	–
.FSB *	105020	–	–	–	–	–
.FWID	–	105301	–	–	–	–
.GOTO	–	–	–	105221	–	–
.IDBL	–	–	–	–	–	105121
.IMAP	105250	–	–	–	–	–
.IMAR	–	–	–	–	105251	105251
.IRES	–	105244	–	–	–	–
.IRT	–	–	–	–	105346	105346
.ISX *	105760	–	–	–	–	–
.ISY *	105770	–	–	–	–	–
.ITBL	–	105122	–	–	–	105122
.JLA *	–	100600	–	–	–	–
.JLB *	–	104600	–	–	–	–
.JLY *	105762	–	–	–	–	–
.JMAP	–	105252	–	–	105252	105252
.JMAR	–	–	–	–	105253	105253
.JPY *	105772	–	–	–	–	–
.JRES	–	105245	–	–	–	–
.LAX *	101742	–	–	–	–	–
.LAY *	101752	–	–	–	–	–
.LBP	–	105257	–	–	105257	105257
.LBPC	–	–	105247	–	–	–
.LBPR	–	105256	–	–	105256	105256
.LBT *	105763	–	–	–	–	–
.LBX *	105742	–	–	–	–	–
.LBY *	105752	–	–	–	–	–
.LDMP	–	105702	–	–	–	–
.LDX *	105745	–	–	–	–	–
.LDY *	105755	–	–	–	–	–
.LFA	–	–	–	101727	–	–

Note: * May be used with or without a leading dot

Table C-1. Instruction Mnemonics in ASCIIbetical Order (sheet 4 of 9)

Mnemonic	HP 1000 Computer Series					
	All 1000s	All A	A990	E,F	E	F
.LLS	–	–	–	–	105347	105347
.LPMR	–	105700	–	–	–	–
.LPX	–	105255	–	–	105255	105255
.LPXR	–	105254	–	–	105254	105254
.LWD1	–	105704	–	–	–	–
.LWD2	–	105705	–	–	–	–
.LXMP	–	–	105712	–	–	–
.LXMR	–	–	105714	–	–	–
.MB00	–	101727	–	–	–	–
.MB01	–	101730	–	–	–	–
.MB02	–	101731	–	–	–	–
.MB10	–	101732	–	–	–	–
.MB11	–	101733	–	–	–	–
.MB12	–	101734	–	–	–	–
.MB20	–	101735	–	–	–	–
.MB21	–	101736	–	–	–	–
.MB22	–	101737	–	–	–	–
.MBF *	–	101732	–	105703	–	–
.MBI *	–	101730	–	105702	–	–
.MBT *	105765	–	–	–	–	–
.MBW *	–	101733	–	105704	–	–
.MPY *	100200	–	–	–	–	–
.MPYD *	–	–	104000	–	–	–
.MVW *	105777	–	–	–	–	–
.MW00	–	105727	–	–	–	–
.MW01	–	105730	–	–	–	–
.MW02	–	105731	–	–	–	–
.MW10	–	105732	–	–	–	–
.MW11	–	105733	–	–	–	–
.MW12	–	105734	–	–	–	–
.MW20	–	105735	–	–	–	–
.MW21	–	105736	–	–	–	–
.MW22	–	105737	–	–	–	–
.MWF *	–	105732	–	105706	–	–
.MWI *	–	105730	–	105705	–	–
.MWW *	–	105733	–	105707	–	–
.NGL	105214	–	–	–	–	–
.PAA	–	–	–	101712	–	–
.PACK	105230	–	–	–	–	–
.PBA	–	–	–	101713	–	–

Note: * May be used with or without a leading dot

Table C-1. Instruction Mnemonics in ASCIIbetical Order (sheet 5 of 9)

Mnemonic	HP 1000 Computer Series					
	All 1000s	All A	A990	E,F	E	F
.PCALI	–	105400	–	–	–	–
.PCALN	–	105404	–	–	–	–
.PCALR	–	105403	–	–	–	–
.PCALV	–	105402	–	–	–	–
.PCALX	–	105401	–	–	–	–
.PMAP	105240	–	–	–	–	–
.PWR2	105225	–	–	–	–	–
.RCS	–	–	105305	–	–	–
.RSA	–	–	–	101730	–	–
.RTC	–	–	105311	–	–	–
.RTM	–	–	105307	–	–	–
.RVA	–	–	–	101731	–	–
.SAX *	101740	–	–	–	–	–
.SAY *	101750	–	–	–	–	–
.SBS *	105773	–	–	–	–	–
.SBT *	105764	–	–	–	–	–
.SBX *	105740	–	–	–	–	–
.SBY *	105750	–	–	–	–	–
.SDSP	–	105405	–	–	–	–
.SETP	105227	–	–	–	–	–
.SFB *	105767	–	–	–	–	–
.SIMP	–	105707	–	–	–	–
.SIP	–	105303	–	–	105350	105350
.SPMR	–	105701	–	–	–	–
.STI0	–	–	–	–	105344	105344
.STMP	–	105703	–	–	–	–
.STX *	105743	–	–	–	–	–
.STY *	105753	–	–	–	–	–
.SWMP	–	105706	–	–	–	–
.SXMP	–	–	105713	–	–	–
.SXMR	–	–	105715	–	–	–
.SYA	–	–	–	101710	–	–
.TADD	–	105002	–	–	–	105002
.TBS *	105775	–	–	–	–	–
.TDIV	–	105062	–	–	–	105062
.TFTD	–	105126	–	–	–	105126
.TFTS	–	105122	–	–	–	105122
.TFXD	–	105106	–	–	–	105106
.TFXS	–	105102	–	–	–	105102
.TICK	–	–	–	–	105342	105342
.TINT	–	105102	–	–	–	105102

Note: * May be used with or without a leading dot

Table C-1. Instruction Mnemonics in ASCIIbetical Order (sheet 6 of 9)

Mnemonic	HP 1000 Computer Series					
	All 1000s	All A	A990	E,F	E	F
.TMPY	–	105042	–	–	–	105042
.TNAM	–	–	–	–	105343	105343
.TPWR	105335	–	–	–	–	–
.TSUB	–	105022	–	–	–	105022
.USA	–	–	–	101711	–	–
.VECT	–	–	–	–	–	101460-000NN0
.WCS	–	–	105304	–	–	–
.WFI	–	105302	–	–	–	–
.WTC	–	–	105310	–	–	–
.WTM	–	–	105306	–	–	–
.XADD	–	–	–	–	105213	105001
.XAX *	101747	–	–	–	–	–
.XAY *	101757	–	–	–	–	–
.XBX *	105747	–	–	–	–	–
.XBY *	105757	–	–	–	–	–
.XCA *	101726	–	–	–	–	–
.XCA1	–	101726	–	–	–	–
.XCA2	–	101723	–	–	–	–
.XCB *	105726	–	–	–	–	–
.XCB1	–	105726	–	–	–	–
.XCB2	–	105723	–	–	–	–
.XCOM	–	–	–	105215	–	–
.XDIV	–	–	–	–	105204	105061
.XFER	105220	–	–	–	–	–
.XFTD	–	–	–	–	–	105125
.XFTS	–	–	–	–	–	105121
.XFXD	–	–	–	–	–	105105
.XFXS	–	–	–	–	–	105101
.XJCQ	–	105711	–	–	–	–
.XJMP	–	105710	–	–	–	–
.XLA	101724	–	–	–	–	–
.XLA1	–	101724	–	–	–	–
.XLA2	–	101721	–	–	–	–
.XLB	105724	–	–	–	–	–
.XLB1	–	105724	–	–	–	–
.XLB2	–	105721	–	–	–	–
.XMPY	–	–	–	–	105203	105041
.XPAK	–	–	–	105206	–	–
.XSA *	101725	–	–	–	–	–
.XSA1	–	101725	–	–	–	–
.XSA2	–	101722	–	–	–	–

Notes: * May be used with or without a leading dot
N Any octal numeric

Table C-1. Instruction Mnemonics in ASCIIbetical Order (sheet 7 of 9)

Mnemonic	HP 1000 Computer Series					
	All 1000s	All A	A990	E,F	E	F
.XSB *	105725	–	–	–	–	–
.XSB1	–	105725	–	–	–	–
.XSB2	–	105722	–	–	–	–
.XSUB	–	–	–	–	105214	105021
.YLD	–	–	–	–	105351	105351
.ZFER	–	105237	–	–	–	–
/ATLG	105333	–	–	–	–	–
/CMRT	105332	–	–	–	–	–
ADA	04(0nn)NNN	–	–	–	–	–
ADA •	14(0nn)NNN	–	–	–	–	–
ADB	04(1nn)NNN	–	–	–	–	–
ADB •	14(1nn)NNN	–	–	–	–	–
ALOG	105322	–	–	–	–	–
ALOGT	105327	–	–	–	–	–
AND	01(0nn)NNN	–	–	–	–	–
AND •	11(0nn)NNN	–	–	–	–	–
ASL	100020...0037	–	–	–	–	–
ASLD	–	–	104040...057	–	–	–
ASR	101020...1037	–	–	–	–	–
ASRD	–	–	104060...077	–	–	–
ATAN	105323	–	–	–	–	–
CLC	10(01n)7NN	–	–	–	–	–
CLF	10(n11)1NN	–	–	–	–	–
CLO	103101	–	–	–	–	–
COS	105324	–	–	–	–	–
CPA	05(0nn)NNN	–	–	–	–	–
CPA •	15(0nn)NNN	–	–	–	–	–
CPB	05(1nn)NNN	–	–	–	–	–
CPB •	15(1nn)NNN	–	–	–	–	–
DBLE	–	–	–	105201	–	–
DDINT	–	–	–	105217	–	–
DPOLY	105331	–	–	–	–	–
DVABS	–	105123	–	–	–	105462
DVADD	–	105021	–	–	–	105460-000000
DVDIV	–	105025	–	–	–	105460-000060
DVDOT	–	105130	–	–	–	105465
DVMAB	–	105132	–	–	–	105467
DVMAX	–	105131	–	–	–	105466
DVMIB	–	105135	–	–	–	105471
DVMIN	–	105133	–	–	–	105470
DVMOV	–	105136	–	–	–	105472
DVMPY	–	105024	–	–	–	105460-000040

Notes: * May be used with or without a leading dot
• Indirect addressing
n Any binary numeric
N Any octal numeric

Table C-1. Instruction Mnemonics in ASCIIbetical Order (sheet 8 of 9)

Mnemonic	HP 1000 Computer Series					
	All 1000s	All A	A990	E,F	E	F
DVNRM	–	105127	–	–	–	105464
DVPIV	–	105121	–	–	–	105461
DVSAD	–	105026	–	–	–	105460-000400
DVSDV	–	105031	–	–	–	105460-000460
DVSMY	–	105030	–	–	–	105460-000440
DVSSB	–	105027	–	–	–	105460-000420
DVSUB	–	105023	–	–	–	105460-000020
DVSUM	–	105125	–	–	–	105463
DVSWP	–	105137	–	–	–	105473
EXP	105326	–	–	–	–	–
FLOAT	105120	–	–	–	–	–
HLT	10(n1n)0NN	–	–	–	–	–
IFIX	105100	–	–	–	–	–
IOR	03(0nn)NNN	–	–	–	–	–
IOR •	13(0nn)NNN	–	–	–	–	–
ISZ	03(1nn)NNN	–	–	–	–	–
ISZ •	13(1nn)NNN	–	–	–	–	–
JMP	02(1nn)NNN	–	–	–	–	–
JMP •	12(1nn)NNN	–	–	–	–	–
JSB	01(1nn)NNN	–	–	–	–	–
JSB •	11(1nn)NNN	–	–	–	–	–
LDA	06(0nn)NNN	–	–	–	–	–
LDA •	16(0nn)NNN	–	–	–	–	–
LDB	06(1nn)NNN	–	–	–	–	–
LDB •	16(1nn)NNN	–	–	–	–	–
LIA	1025NN	–	–	–	–	–
LIAC	1035NN	–	–	–	–	–
LIB	1065NN	–	–	–	–	–
LIBC	1075NN	–	–	–	–	–
LSL	100040...0057	–	–	–	–	–
LSLD	–	–	100060...077	–	–	–
LSR	101040...1057	–	–	–	–	–
LSRD	–	–	101060...077	–	–	–
MIA	1024NN	–	–	–	–	–
MIAC	1034NN	–	–	–	–	–
MIB	1064NN	–	–	–	–	–
MIBC	1074NN	–	–	–	–	–
OTA	10(01n)6NN	–	–	–	–	–
OTB	10(11n)6NN	–	–	–	–	–
RRL	100100...0117	–	–	–	–	–
RRR	101100...1117	–	–	–	–	–
SFC	10(n1n)2NN	–	–	–	–	–
SFS	10(n1n)3NN	–	–	–	–	–

Notes: • Indirect addressing
n Any binary numeric
N Any octal numeric

Table C-1. Instruction Mnemonics in ASCIIbetical Order (sheet 9 of 9)

Mnemonic	HP 1000 Computer Series					
	All 1000s	All A	A990	E,F	E	F
SIN	105325	–	–	–	–	–
SNGL	–	–	–	105202	–	–
SOC	102201	–	–	–	–	–
SOCC	103201	–	–	–	–	–
SOS	102301	–	–	–	–	–
SOSC	103301	–	–	–	–	–
SQRT	105321	–	–	–	–	–
STA	07(0nn)NNN	–	–	–	–	–
STA •	17(0nn)NNN	–	–	–	–	–
STB	07(1nn)NNN	–	–	–	–	–
STB •	17(1nn)NNN	–	–	–	–	–
STC	10(11n)7NN	–	–	–	–	–
STF	10(n10)1NN	–	–	–	–	–
STO	102101	–	–	–	–	–
TAN	105320	–	–	–	–	–
TANH	105330	–	–	–	–	–
VABS	–	105103	–	–	–	101462
VADD	–	105001	–	–	–	101460-000000
VDIV	–	105005	–	–	–	101460-000060
VDOT	–	105110	–	–	–	101465
VMAB	–	105112	–	–	–	101467
VMAX	–	105111	–	–	–	101466
VMIB	–	105115	–	–	–	101471
VMIN	–	105113	–	–	–	101470
VMOV	–	105116	–	–	–	101472
VMPY	–	105004	–	–	–	101460-000040
VNRM	–	105107	–	–	–	101464
VPIV	–	105101	–	–	–	101461
XOR	02(0nn)NNN	–	–	–	–	–
VSAD	–	105006	–	–	–	101460-000400
VSDV	–	105011	–	–	–	101460-000460
VSMY	–	105010	–	–	–	101460-000440
VSSB	–	105007	–	–	–	101460-000420
VSUB	–	105003	–	–	–	101460-000020
VSUM	–	105105	–	–	–	101463
VSWP	–	105117	–	–	–	101473
XADD	–	–	–	–	105207	–
XDIV	–	–	–	–	105212	–
XLUEX	–	100710	–	–	–	–
XMA	–	–	–	101722	–	–
XMPY	–	–	–	–	105211	–

Notes: • Indirect addressing
n Any binary numeric
N Any octal numeric

Instruction Mnemonics in Opcode (Octal) Order

In *general,* A-Series computer (CPU and SPU) instructions which are not implemented (denoted with a dash in Table C-2) will cause an unimplemented instruction trap when executed. Note that RTE-A uses the following unimplemented instructions as traps:

EXEC	100700
$LIBR	100701
$LIBX	100702
$SJP	100703
$SJSO	100704
$SJSI	100705
$SJS2	100706
$SJS3	100707
XLUEX	100710
$PRIV	100711

Most unimplemented instructions on the E/F-Series computers are no-ops. Some, however, will produce unpredictable results while others may cause the computer to hang. None of the unimplemented instructions are useful.

Note The instruction set listed in each computer series reference manual takes precedence over Table C-2 and, therefore, should be considered as the ultimate source for instruction sets.

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 1 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
01(0nn)NNN	AND	AND	AND	AND	AND	AND	AND	AND
01(1nn)NNN	JSB	JSB	JSB	JSB	JSB	JSB	JSB	JSB
02(0nn)NNN	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
02(1nn)NNN	JMP	JMP	JMP	JMP	JMP	JMP	JMP	JMP
03(0nn)NNN	IOR	IOR	IOR	IOR	IOR	IOR	IOR	IOR
03(1nn)NNN	ISZ	ISZ	ISZ	ISZ	ISZ	ISZ	ISZ	ISZ
04(0nn)NNN	ADA	ADA	ADA	ADA	ADA	ADA	ADA	ADA
04(1nn)NNN	ADB	ADB	ADB	ADB	ADB	ADB	ADB	ADB
05(0nn)NNN	CPA	CPA	CPA	CPA	CPA	CPA	CPA	CPA
05(1nn)NNN	CPB	CPB	CPB	CPB	CPB	CPB	CPB	CPB
06(0nn)NNN	LDA	LDA	LDA	LDA	LDA	LDA	LDA	LDA
06(1nn)NNN	LDB	LDB	LDB	LDB	LDB	LDB	LDB	LDB
07(0nn)NNN	STA	STA	STA	STA	STA	STA	STA	STA
07(1nn)NNN	STB	STB	STB	STB	STB	STB	STB	STB
100000...017	–	–	–	–	–	–	–	–
100020...037	ASL	ASL	ASL	ASL	ASL	ASL	ASL	ASL
100040...057	LSL	LSL	LSL	LSL	LSL	LSL	LSL	LSL
100060...077	–	–		–	–	LSLD	–	–
100100...117	RRL	RRL	RRL	RRL	RRL	RRL	RRL	RRL
100120...177	–	–	–	–	–	–	–	–
100200	MPY *	MPY *	MPY *	MPY *	MPY *	MPY *	MPY *	MPY *
100201...377	–	–	–	–	–	–	–	–
100400	DIV *	DIV *	DIV *	DIV *	DIV *	DIV *	DIV *	DIV *
100401...577	–	–	–	–	–	–	–	–
100600	JLA *	JLA *	JLA *	JLA *	JLA *	JLA *	–	–
100601...777	–	–	–	–	–	–	–	–
10(01n)7NN	CLC	CLC	CLC	CLC	CLC	CLC	CLC	CLC
10(n11)1NN	CLF	CLF	CLF	CLF	CLF	CLF	CLF	CLF
10(01n)6NN	OTA	OTA	OTA	OTA	OTA	OTA	OTA	OTA
10(11n)6NN	OTB	OTB	OTB	OTB	OTB	OTB	OTB	OTB
10(n10)1NN	STF	STF	STF	STF	STF	STF	STF	STF
10(11n)7NN	STC	STC	STC	STC	STC	STC	STC	STC
101000...017	–	–	–	–	–	–	–	–
101020...037	ASR	ASR	ASR	ASR	ASR	ASR	ASR	ASR
101040...057	LSR	LSR	LSR	LSR	LSR	LSR	LSR	LSR
101060...077	–	–	–	–	–	LSRD	–	–
101100...117	RRR	RRR	RRR	RRR	RRR	RRR	RRR	RRR
101120...405	–	–	–	–	–	–	–	–
101406	.CCQA	–	.CCQA	.CCQA	.CCQA	.CCQA	–	–

Notes: * May be used with or without a leading dot
n Any binary numeric
N Any octal numeric

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 2 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
101407	.CACQ	–	.CACQ	.CACQ	.CACQ	.CACQ	–	–
101410	.CZA	–	.CZA	.CZA	.CZA	.CZA	–	–
101411	.CAZ	–	.CAZ	.CAZ	.CAZ	.CAZ	–	–
101412	.CIQA	–	.CIQA	.CIQA	.CIQA	.CIQA	–	–
101413	.ADQA	–	.ADQA	.ADQA	.ADQA	.ADQA	–	–
101414...457	–	–	–	–	–	–	–	–
101460–	–	–	–	–	–	–	–	.VECT
–000000	–	–	–	–	–	–	–	VADD
–000020	–	–	–	–	–	–	–	VSUB
–000040	–	–	–	–	–	–	–	VMPY
–000060	–	–	–	–	–	–	–	VDIV
–000400	–	–	–	–	–	–	–	VSAD
–000420	–	–	–	–	–	–	–	VSSB
–000440	–	–	–	–	–	–	–	VSMY
–000460	–	–	–	–	–	–	–	VSDV
101461	–	–	–	–	–	–	–	VPIV
101462	–	–	–	–	–	–	–	VABS
101463	–	–	–	–	–	–	–	VSUM
101464	–	–	–	–	–	–	–	VNRM
101465	–	–	–	–	–	–	–	VDOT
101466	–	–	–	–	–	–	–	VMAX
101467	–	–	–	–	–	–	–	VMAB
101470	–	–	–	–	–	–	–	VMIN
101471	–	–	–	–	–	–	–	VMIB
101472	–	–	–	–	–	–	–	VMOV
101473	–	–	–	–	–	–	–	VSWP
101474...677	–	–	–	–	–	–	–	–
101700...707	–	–	–	–	–	–	–	–
101010	–	–	–	–	–	–	.SYA	.SYA
101711	–	–	–	–	–	–	.USA	.USA
101712	–	–	–	–	–	–	.PAA	.PAA
101713	–	–	–	–	–	–	.PBA	.PBA
101714...715	–	–	–	–	–	–	–	–
101716	–	–	–	–	.XLAB	–	–	–
101717...720	–	–	–	–	–	–	–	–
101721	.XLA2	.XLA2	.XLA2	.XLA2	.XLA2	.XLA2	–	–
101722	.XSA2	.XSA2	.XSA2	.XSA2	.XSA2	.XSA2	XMA	XMA
101723	.XCA2	.XCA2	.XCA2	.XCA2	.XCA2	.XCA2	–	–
101724	.XLA1	.XLA1	.XLA1	.XLA1	.XLA1	.XLA1	XLA *	XLA *
101725	.XSA1	.XSA1	.XSA1	.XSA1	.XSA1	.XSA1	XSA *	XSA *
101726	.XCA1	.XCA1	.XCA1	.XCA1	.XCA1	.XCA1	XCA *	XCA *
101727	.MB00	.MB00	.MB00	.MB00	.MB00	.MB00	.LFA	.LFA
101730	.MB01	.MB01	.MB01	.MB01	.MB01	.MB01	.RSA	.RSA

Note: * May be used with or without a leading dot

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 3 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
101731	.MB02	.MB02	.MB02	.MB02	.MB02	.MB02	.RVA	.RVA
101732	.MB10	.MB10	.MB10	.MB10	.MB10	.MB10	–	–
101733	.MB11	.MB11	.MB11	.MB11	.MB11	.MB11	–	–
101734	.MB12	.MB12	.MB12	.MB12	.MB12	.MB12	–	–
101735	.MB20	.MB20	.MB20	.MB20	.MB20	.MB20	–	–
101736	.MB21	.MB21	.MB21	.MB21	.MB21	.MB21	–	–
101737	.MB22	.MB22	.MB22	.MB22	.MB22	.MB22	–	–
101740	SAX *	SAX *	SAX *	SAX *	SAX *	SAX *	SAX *	SAX *
101741	CAX *	CAX *	CAX *	CAX *	CAX *	CAX *	CAX *	CAX *
101742	LAX *	LAX *	LAX *	LAX *	LAX *	LAX *	LAX *	LAX *
101743	–	–	–	–	–	–	–	–
101744	CXA *	CXA *	CXA *	CXA *	CXA *	CXA *	CXA *	CXA *
101745	–	–	–	–	–	–	–	–
101746	–	–	–	–	–	–	–	–
101747	XAX *	XAX *	XAX *	XAX *	XAX *	XAX *	XAX *	XAX *
101750	SAY *	SAY *	SAY *	SAY *	SAY *	SAY *	SAY *	SAY *
101751	CAY *	CAY *	CAY *	CAY *	CAY *	CAY *	CAY *	CAY *
101752	LAY *	LAY *	LAY *	LAY *	LAY *	LAY *	LAY *	LAY *
101753	–	–	–	–	–	–	–	–
101754	CYA *	CYA *	CYA *	CYA *	CYA *	CYA *	CYA *	CYA *
101755	–	–	–	–	–	–	–	–
101756	–	–	–	–	–	–	–	–
101757	XAY *	XAY *	XAY *	XAY *	XAY *	XAY *	XAY *	XAY *
101760...777	–	–	–	–	–	–	–	–
102101	STO	STO	STO	STO	STO	STO	STO	STO
102201	SOC	SOC	SOC	SOC	SOC	SOC	SOC	SOC
102301	SOS	SOS	SOS	SOS	SOS	SOS	SOS	SOS
1024NN	MIA	MIA	MIA	MIA	MIA	MIA	MIA	MIA
1025NN	LIA	LIA	LIA	LIA	LIA	LIA	LIA	LIA
103101	CLO	CLO	CLO	CLO	CLO	CLO	CLO	CLO
104000	–	–	–	–	–	.MPYD	–	–
104001...037	–	–	–	–	–	–	–	–
104040...057	–	–	–	–	–	ASLD	–	–
104060...077	–	–	–	–	–	ASRD	–	–
104100	–	–	–	–	–	.DIVD	–	–
104101...177	–	–	–	–	–	–	–	–
104200	DLD *	DLD *	DLD *	DLD *	DLD *	DLD *	DLD *	DLD *
104201...377	–	–	–	–	–	–	–	–
104400	DST *	DST *	DST *	DST *	DST *	DST *	DST *	DST *
104401...577	–	–	–	–	–	–	–	–

Note: * May be used with or without a leading dot

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 4 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
104600	JLB *	JLB *	JLB *	JLB *	JLB *	JLB *		
104601.....777	–	–	–	–	–	–	–	–
105000	FAD *	FAD *	FAD *	FAD *	FAD *	FAD *	FAD *	FAD *
105001	–	–	–	VADD‡	VADD	VADD	–	.XADD
105002		–		.TADD‡	.TADD	.TADD	–	.TADD
105003	–	–	–	VSUB‡	VSUB	VSUB	–	–
105004	–	–	–	VMPY‡	VMPY	VMPY	–	–
105005	–	–	–	VDIV‡	VDIV	VDIV	–	–
105006	–	–	–	VSAD‡	VSAD	VSAD	–	–
105007	–	–	–	VSSB	VSSB	VSSB	–	–
105010	–	–	–	VSMY‡	VSMY	VSMY	–	–
105011	–	–	–	VSDV‡	VSDV	VSDV	–	–
105012	–	–	–	–	–	–	–	–
105013	–	–	–	–	–	–	–	–
105014	.DAD	.DAD	.DAD	.DAD	.DAD	.DAD	–	.DAD†
105015	–	–	–	–	–	–	–	–
105016	–	–	–	–	–	–	–	–
105017	–	–	–	–	–	–	–	–
105020	FSB *	FSB *	FSB *	FSB *	FSB *	FSB *	FSB *	FSB *
105021	–	–	–	DVADD‡	DVADD	DVADD	–	.XSUB
105022	.TSUB	–	.TSUB	.TSUB‡	.TSUB	.TSUB	–	.TSUB
105023	–	–	–	DVSUB‡	DVSUB	DVSUB	–	–
105024	–	–	–	DVMPY‡	DVMPY	DVMPY	–	–
105025	–	–	–	DVDIV‡	DVDIV	DVDIV	–	–
105026	–	–	–	DVSAD‡	DVSAD	DVSAD	–	–
105027	–	–	–	DVSSB‡	DVSSB	DVSSB	–	–
105030	–	–	–	DVSMY‡	DVSMY	DVSMY	–	–
105031	–	–	–	DVSDV‡	DVSDV	DVSDV	–	–
105032	–	–	–	–	–	–	–	–
105033	–	–	–	–	–	–	–	–
105034	.DSB	.DSB	.DSB	.DSB	.DSB	.DSB	–	.DSB†
105035	–	–	–	–	–	–	–	–
105036	–	–	–	–	–	–	–	–
105037	–	–	–	–	–	–	–	–
105040	FMP *	FMP *	FMP *	FMP *	FMP *	FMP *	FMP *	FMP *
105041	–	–	–	–	–	–	–	.XMPY
105042	.TMPY	–	.TMPY	.TMPY‡	.TMPY	.TMPY	–	.TMPY
105043	–	–	–	–	–	–	–	–
105044	–	–	–	–	–	–	–	–
105045	–	–	–	–	–	–	–	–
105046	–	–	–	–	–	–	–	–

Notes:
* May be used with or without a leading dot
† F-Series with date code later than 1920 (firmware revision $\geq$ 2)
‡ A700 Series with hardware floating point

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 5 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
105047	–	–	–	–	–	–	–	–
105050	–	–	–	–	–	–	–	–
105051	–	–	–	–	–	–	–	–
105052	–	–	–	–	–	–	–	–
105053	–	–	–	–	–	–	–	–
105054	.DMP	–	.DMP	.DMP	.DMP	.DMP	–	.DMP†
105055	–	–	–	–	–	–	–	–
105056	–	–	–	–	–	–	–	–
105057	–	–	–	–	–	–	–	–
105060	FDV *	FDV *	FDV *	FDV *	FDV *	FDV *	FDV *	FDV *
105061	–	–	–	–	–	–	–	.XDIV
105062	.TDIV	–	.TDIV	.TDIV‡	.TDIV	.TDIV	–	.TVID
105063	–	–	–	–	–	–	–	–
105064	–	–	–	–	–	–	–	–
105065	–	–	–	–	–	–	–	–
105066	–	–	–	–	–	–	–	–
105067	–	–	–	–	–	–	–	–
105070	–	–	–	–	–	–	–	–
105071	–	–	–	–	–	–	–	–
105072	–	–	–	–	–	–	–	–
105073	–	–	–	–	–	–	–	–
105074	.DDI	–	.DDI	.DDI	.DDI	.DDI	–	DDI†
105075	–	–	–	–	–	–	–	–
105076	–	–	–	–	–	–	–	–
105077	–	–	–	–	–	–	–	–
105100	FIX *	FIX *	FIX *	FIX *	FIX *	FIX *	FIX *	FIX *
105101	–	–	–	VPIV‡	VPIV	VPIV	–	.XFXS
							–	.DINT
105102	.TINT	–	.TINT	.TINT	.TINT	.TINT	–	.TINT
	.TFXS	–	.TFXS	.TFXS‡	.TFXS	.TFXS	–	.TFXS
105103	–	–	–	VABS‡	VABS	VABS	–	–
105104	.FIXD	–	.FIXD	.FIXD‡	.FIXD	.FIXD	–	.FIXD
105105	–	–	–	VSUM‡	VSUM	VSUM	–	.XFXD
105106	.TFXD	–	.TFXD	.TFXD‡	.TFXD	.TFXD	–	.TFXD
105107	–	–	–	VNRM‡	VNRM	VNRM	–	–
105110	–	–	–	VDOT‡	VDOT	VDOT	–	–
105111	–	–	–	VMAX‡	VMAX	VMAX	–	–
105112	–	–	–	VMAB‡	VMAB	VMAB	–	–
105113	–	–	–	VMIN‡	VMIN	VMIN	–	–
105114	.DSBR	.DSBR	.DSBR	.DSBR	.DSBR	.DSBR	–	.DSBR†
105115	–	–	–	VMIB‡	VMIB	VMIB	–	–
105116	–	–	–	VMOV‡	VMOV	VMOV	–	–

Notes: * May be used with or without a leading dot
† F-Series with date code later than 1920 (firmware revision $\geq$ 2)
‡ A700 Series with hardware floating point

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 6 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
105117	–	–	–	VSWP‡	VSWP	VSWP	–	–
105120	FLT *	FLT *	FLT *	FLT *	FLT *	FLT *	FLT *	FLT *
105121	–	–	–	DVPIV‡	DVPIV	DVPIV	–	.XFTS
							–	.IDBL
105122	.ITBL	–	.ITBL	.ITBL‡	.ITBL	.ITBL	–	.ITBL
	.TFTS	–	.TFTS	.TFTS‡	.TFTS	.TFTS	–	.TFTS
105123	–	–	–	DVABS‡	DVABS	DVABS	–	–
105124	.FLTD	–	.FLTD	.FLTD‡	.FLTD	.FLTD	–	.FLTD
105125	–	–	–	DVSUM‡	DVSUM	DVSUM	–	.XFTD
105126	.TFTD	–	.TFTD	.TFTD‡	.TFTD	.TFTD	–	.TFTD
105127	–	–	–	DVNRM‡	DVNRM	DVNRM	–	–
105130	–	–	–	DVDOT‡	DVDOT	DVDOT	–	–
105131	–	–	–	DVMAX‡	DVMAX	DVMAX	–	–
105132	–	–	–	DVMAB‡	DVMAB	DVMAB	–	–
105133	–	–	–	DVMIN‡	DVMIN	DVMIN	–	–
105134	.DDIR	–	.DDIR	.DDIR	.DDIR	.DDIR	–	.DDIR†
105135	–	–	–	DVMIB‡	DVMIB	DVMIB	–	–
105136	–	–	–	DVMOV‡	DVMOV	DVMOV	–	–
105137	–	–	–	DVSWP‡	DVSWP	DVSWP	–	–
105140...177	–	–	–	–	–	–	–	–
105200	–	–	–	–	–	–	–	–
105201	–	–	–	–	–	–	DBLE	DBLE
105202	–	–	–	–	–	–	SNGL	SNGL
105203	.DNG	.DNG	.DNG	.DNG	.DNG	.DNG	.XMPY	.DNG†
105204	.DCO	.DCO	.DCO	.DCO	.DCO	.DCO	.XDIV	.DCO†
105205	.DFER	.DFER	.DFER	.DFER	.DFER	.DFER	.DFER	.DFER
105206	–	–	–	–	–	–	.XPAK	.XPAK
105207	.BLE	–	.BLE	.BLE‡	.BLE	.BLE	XADD	.BLE†
105210	.DIN	.DIN	.DIN	.DIN	.DIN	.DIN	.XSUB	.DIN†
105211	.DDE	.DDE	.DDE	.DDE	.DDE	.DDE	XMPY	.DDE†
105212	.DIS	.DIS	.DIS	.DIS	.DIS	.DIS	XDIV	.DIS†
105213	.DDS	.DDS	.DDS	.DDS	.DDS	.DDS	.XADD	.DDS†
105214	.NGL	–	.NGL	.NGL‡	.NGL	.NGL	.XSUB	.NGL†
105215	–	–	–	–	–	–	.XCOM	.XCOM
105216	–	–	–	–	–	–	..DCM	..DCM
105217	–	–	–	–	–	–	DDINT	DDINT
105220	.XFER	.XFER	.XFER	.XFER	.XFER	.XFER	.XFER	.XFER
105221	–	–	–	–	–	–	.GOTO	.GOTO
105222	–	–	–	–	–	.DSZ	..MAP	..MAP
105223	.ENTR	.ENTR	.ENTR	.ENTR	.ENTR	.ENTR	.ENTR	.ENTR
105224	.ENTP	.ENTP	.ENTP	.ENTP	.ENTP	.ENTP	.ENTP	.ENTP
105225	.PWR2	–	.PWR2	.PWR2	.PWR2	.PWR2	.PWR2	.PWR2

Notes:
* May be used with or without a leading dot
† F-Series with date code later than 1920 (firmware revision ≥ 2)
‡ A700 Series with hardware floating point

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 7 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
105226	.FLUN	–	.FLUN	.FLUN	.FLUN	.FLUN	.FLUN	.FLUN
105227	.SETP	.SETP	.SETP	.SETP	.SETP	.SETP	$SETP	$SETP
105230	.PACK		.PACK	.PACK	.PACK	.PACK	.PACK	.PACK
105231	.CFER	.CFER	.CFER	.CFER	.CFER	.CFER	.CFER	.CFER
105232	..FCM	..FCM	..FCM	..FCM	..FCM	..FCM	–	..FCM†
105233	..TCM	–	..TCM	..TCM‡	..TCM	..TCM	–	..TCM†
105234	.ENTN	.ENTN	.ENTN	.ENTN	.ENTN	.ENTN	–	–
105235	.ENTC	.ENTC	.ENTC	.ENTC	.ENTC	.ENTC	–	–
105236	.CPM	.CPM	.CPM	.CPM	.CPM	.CPM	–	–
105237	.ZFER	.ZFER	.ZFER	.ZFER	.ZFER	.ZFER		–
105240	.PMAP	.PMAP	.PMAP	.PMAP	.PMAP	.PMAP	.PMAP	.PMAP
105241	–	–	–	–	–	–	$LOC	$LOC
105242	–	–	–	–	–	–	–	–
105243	–	–	–	–	–	–	–	–
105244	.IRES	.IRES	.IRES	.IRES	.IRES	.IRES	–	–
105245	.JRES	–	.JRES	.JRES	.JRES	.JRES	–	–
105246	–	–	–	–	–	–	–	–
105247	–	–	–	–	–	.LBPC	–	–
105250	.IMAP	.IMAP	.IMAP	.IMAP	.IMAP	.IMAP	.IMAP	.IMAP
105251	–	–	–	–	–	–	.IMAR	.IMAR
105252	.JMAP	–	.JMAP	.JMAP	.JMAP	.JMAP	.JMAP	.JMAP
105253	–	–	–	–	–	–	.JMAR	.JMAR
105254	.LPXR	.LPXR	.LPXR	.LPXR	.LPXR	.LPXR	.LPXR	.LPXR
105255	.LPX	.LPX	.LPX	.LPX	.LPX	.LPXR	.LPX	.LPX
105256	.LBPR	.LBPR	.LBPR	.LBPR	.LBPR	.LBPR	.LBPR	.LBPR
105257	.LBP	.LBP	.LBP	.LBP	.LBP	.LBP	.LBP	.LBP
105260	–	–	–	–	–	–	–	–
105300	.CPUID	.CPUID	.CPUID	.CPUID	.CPUID	.CPUID	.CPUID	.CPUID
	.CPU	.CPU	.CPU	.CPU	.CPU	.CPU	–	–
105301	.FWID	.FWID	.FWID	.FWID	.FWID	.FWID	.FWID	.FWID
105302	.WFI	.WFI	.WFI	.WFI	.WFI	.WFI	–	–
105303	.SIP	.SIP	.SIP.	SIP	.SIP	.SIP	–	–
105304	–	–	–	–	–	.WCS	–	–
105305	–	–	–	–	–	.RCS	–	–
105306	–	–	–	–	–	.WTM	–	–
105307	–	–	–	–	–	.RTM	–	–
105310	–	–	–	–	–	.WTC	–	–
105311	–	–	–	–	–	.RTC	–	–
105312	–	–	–	–	–	–	–	–
105313	–	–	–	–	–	–	–	–
105314	–	–	–	–	–	–	–	–
105315	–	–	–	–	–	–	–	–

Note: † F-Series with date code later than 1920 (firmware revision ≥ 2)
‡ A700 Series with hardware floating point

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 8 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
105316	–	–	–	–	–	–	–	–
105317	–	–	–	–	–	–	–	–
105320	–	–	–	TAN‡	TAN	TAN	self-test	TAN
105321	–	–	–	SQRT‡	SQRT	SQRT	.DAD	SQRT
105322	–	–	–	ALOG‡	ALOG	ALOG	.DMP	ALOG
105323	–	–	–	ATAN‡	ATAN	ATAN	.DNG	ATAN
105324	–	–	–	COS‡	COS	COS	.DCO	COS
105325	–	–	–	SIN‡	SIN	SIN	.DDI	SIN
105326	–	–	–	EXP‡	EXP	EXP	.DDIR	EXP
105327	–	–	–	ALOGT‡	ALOGT	ALOGT	.DSB	ALOGT
105330	–	–	–	TANH‡	TANH	TANH	.DIN	TANH
105331	–	–	–	DPOLY‡	DPOLY	DPOLY	.DDE	DPOLY†
105332	–	–	–	/CMRT‡	/CMRT	/CMRT	.DIS	/CMRT†
105333	–	–	–	/ATLG‡	/ATLG	/ATLG	..DDS	/ATLG†
105334	–	–	–	.FRWR‡	.FPWR	.FPWR	.DSBR	.FPWR†
105335	–	–	–	.TPWR‡	.TPWR	.TPWR	–	.TPWR†
105336	–	–	–	–	–	–	–	–
105337	–	–	–	–	–	–	–	–
105340	–	–	–	–	–	–	$LIBR	$LIBR
105341	–	–	–	–	–	–	$LIBX	$LIBX
105342	–	–	–	–	–	–	.TICK	.TICK
105343	–	–	–	–	–	–	.TNAM	.TNAM
105344	–	–	–	–	–	–	.STIO	.STIO
105345	–	–	–	–	–	–	.FNW	.FNW
105346	–	–	–	–	–	–	.IRT	.IRT
105347	–	–	–	–	–	–	.LLS	.LLS
105350	–	–	–	–	–	–	.SIP	.SIP
105351	–	–	–	–	–	–	.YLD	.YLD
105352	–	–	–	–	–	–	.CPM	.CPM
105353	–	–	–	–	–	–	.ETEQ	.ETEQ
105354	–	–	–	–	–	–	.ENTN	.ENTN
105355	–	–	–	–	–	–	self-test	self-test
105356	–	–	–	–	–	–	.ENTC	.ENTC
105357	–	–	–	–	–	–	.DSPI	.DSPI
105360...377	–	–	–	–	–	–	–	–
105400	.PCALI	–	.PCALI	.PCALI	.PCALI	.PCALI	–	–
105401	.PCALX	–	.PCALX	.PCALX	.PCALX	.PCALX	–	–
105402	.PCALV	–	.PCALV	.PCALV	.PCALV	.PCALV	–	–
105403	.PCALR	–	.PCALR	.PCALR	.PCALR	.PCALR	–	–
105404	.PCALN	–	.PCALN	.PCALN	.PCALN	.PCALN	–	–
105405	.SDSP	–	.SDSP	.SDSP	.SDSP	.SDSP	–	–
105406	.CCQB	–	.CCQB	.CCQB	.CCQB	.CCQB	–	–

Notes: † F-Series with date code later than 1920 (firmware revision $\geq$ 2)
‡ A700 Series with hardware floating point

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 9 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
105407	.CBCQ	–	.CBCQ	.CBCQ	.CBCQ	.CBCQ	–	–
105410	.CZB	–	.CZB	.CZB	.CZB	.CZB	–	–
105411	.CBZ	–	.CBZ	.CBZ	.CBZ	.CBZ	–	–
105412	.CIQB	–	.CIQB	.CIQB	.CIQB	.CIQB	–	–
105413	.ADQB	–	.ADQB	.ADQB	.ADQB	.ADQB	–	–
105414	–	–	–	–	–	–	–	–
105415	.EXIT1	–	.EXIT1	.EXIT1	.EXIT1	.EXIT1	–	–
105416	.EXIT2	–	.EXIT2	.EXIT2	.EXIT2	.EXIT2	–	–
105417	.EXIT0	–	.EXIT0	.EXIT0	.EXIT0	.EXIT0	–	–
105420...457	–	–	–	–	–	–	–	–
105460–	–	–	–	–	–	–	–	.DVCT
–000000	–	–	–	–	–	–	–	DVADD
–000020	–	–	–	–	–	–	–	DVSUB
–000040	–	–	–	–	–	–	–	DVMPY
–000060	–	–	–	–	–	–	–	DVDIV
–000400	–	–	–	–	–	–	–	DVSAD
–000420	–	–	–	–	–	–	–	DVSSB
–000440	–	–	–	–	–	–	–	DVSMY
–000460	–	–	–	–	–	–	–	DVSDV
–105461	–	–	–	–	–	–	–	DVPIV
–105462	–	–	–	–	–	–	–	DVABS
–105463	–	–	–	–	–	–	–	DVSUM
–105464	–	–	–	–	–	–	–	DVNRM
–105465	–	–	–	–	–	–	–	DVDOT
105466	–	–	–	–	–	–	–	DVMAX
105467	–	–	–	–	–	–	–	DVMAB
105470	–	–	–	–	–	–	–	DVMIN
105471	–	–	–	–	–	–	–	DVMIB
105472	–	–	–	–	–	–	–	DVMOV
105473	–	–	–	–	–	–	–	DVSWP
105474...477	–	–	–	–	–	–	–	–
105500...677	–	–	–	–	–	–	–	–
105700	.LPMR	.LPMR	.LPMR	.LPMR	.LPMR	.LPMR	–	–
105701	.SPMR	.SPMR	.SPMR	.SPMR	.SPMR	.SPMR	–	–
105702	.LDMP	.LDMP	.LDMP	.LDMP	.LDMP	.LDMP	MBI *	MBI *
105703	.STMP	.STMP	.STMP	.STMP	.STMP	.STMP	MBF *	MBF *
105704	.LWD1	.LWD1	.LWD1	.LWD1	.LWD1	.LWD1	MBW *	MBW *
105705	.LWD2	.LWD2	.LWD2	.LWD2	.LWD2	.LWD2	MWI *	MWI *
105706	.SWMP	.SWMP	.SWMP	.SWMP	.SWMP	.SWMP	MWF *	MWF *
105707	.SIMP	.SIMP	.SIMP	.SIMP	.SIMP	.SIMP	MWW *	MWW *
105710	.XJMP	.XJMP	.XJMP	.XJMP	.XJMP	.XJMP	SYB	SYB
105711	.XJCQ	.XJCQ	.XJCQ	.XJCQ	.XJCQ	.XJCQ	USB	USB

Note: * May be used with or without a leading dot

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 10 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
105712	–	–	–	–	–	.LXMP	PAB	PAB
105713	–	–	–	–	–	.SXMP	PBB	PBB
105714	–	–	–	–	–	.LXMR	SSM	SSM
105715	–	–	–	–	–	.SXMR	JRS	JRS
105716	–	–	–	–	.XLBB	–	–	–
105717	–	–	–	–	–	–	–	–
105720	–	–	–	–	–	–	XMM	XMM
105721	.XLB2	.XLB2	.XLB2	.XLB2	.XLB2	.XLB2	XMS	XMS
105722	.XSB2	.XSB2	.XSB2	.XSB2	.XSB2	.XSB2	XMB	XMB
105723	.XCB2	.XCB2	.XCB2	.XCB2	.XCB2	.XCB2	–	–
105724	.XLB1	.XLB1	.XLB1	.XLB1	.XLB1	.XLB1	XLB *	XLB *
105725	.XSB1	.XSB1	.XSB1	.XSB1	.XSB1	.XSB1	XSB *	XSB *
105726	.XCB1	.XCB1	.XCB1	.XCB1	.XCB1	.XCB1	XCB *	XCB *
105727	.MW00	.MW00	.MW00	.MW00	.MW00	.MW00	LFB	LFB
105730	.MW01	.MW01	.MW01	.MW01	.MW01	.MW01	RSB	RSB
105731	.MW02	.MW02	.MW02	.MW02	.MW02	.MW02	RVB	RVB
105732	.MW10	.MW10	.MW10	.MW10	.MW10	.MW10	DHP	DHP
105733	.MW11	.MW11	.MW11	.MW11	.MW11	.MW11	DJS	DJS
105734	.MW12	.MW12	.MW12	.MW12	.MW12	.MW12	SJP	SJP
105735	.MW20	.MW20	MW20	.MW20	.MW20	.MW20	SJS	SJS
105736	.MW21	.MW21	.MW21	.MW21	.MW21	.MW21	UJB	UJB
105737	.MW22	.MW22	.MW22	.MW22	.MW22	.MW22	UJS	UJS
105740	SBX *	SBX *	SBX *	SBX *	SBX *	SBX *	SBX *	SBX *
105741	CBX *	CBX *	CBX *	CBX *	CBX *	CBX *	CBX *	CBX *
105742	LBX *	LBX *	LBX *	LBX *	LBX *	LBX *	LBX *	LBX *
105743	STX *	STX *	STX *	STX *	STX *	STX *	STX *	STX *
105744	CXB *	CXB *	CXB *	CXB *	CXB *	CXB *	CXB *	CXB *
105745	LDX *	LDX *	LDX *	LDX *	LDX *	LDX *	LDX *	LDX *
105746	ADX *	ADX *	ADX *	ADX *	ADX *	ADX *	ADX *	ADX *
105747	XBX *	XBX *	XBX *	XBX *	XBX *	XBX *	XBX *	XBX *
105750	SBY *	SBY *	SBY *	SBY *	SBY	SBY *	SBY *	SBY *
105751	CBY *	CBY *	CBY *	CBY *	CBY *	CBY *	CBY *	CBY *
105752	LBY *	LBY *	LBY *	LBY *	LBY *	LBY *	LBY *	LBY *
105753	STY *	STY *	STY *	STY *	STY *	STY *	STY *	STY *
105754	CYB *	CYB *	CYB *	CYB *	CYB *	CYB *	CYB *	CYB *
105755	LDY *	LDY *	LDY *	LDY *	LDY *	LDY *	LDY *	LDY *
105756	ADY *	ADY *	ADY *	ADY *	ADY *	ADY *	ADY *	ADY *
105757	XBY *	XBY *	XBY *	XBY *	XBY *	XBY *	XBY *	XBY *
105760	ISX *	ISX *	ISX *	ISX *	ISX *	ISX *	ISX *	ISX *

Note: * May be used with or without a leading dot

Table C-2. Instruction Mnemonics in Opcode (Octal) Order (sheet 11 of 11)

OPCODE (OCTAL)	HP 1000 Computer Series							
	A400	A600	A600+	A700	A900	A990	E	F
105761	DSX *	DSX *	DSX *	DSX *	DSX *	DSX *	DSX *	DSX *
105762	JLY *	JLY *	JLY *	JLY *	JLY *	JLY *	JLY *	JLY *
105763	LBT *	LBT *	LBT *	LBT *	LBT *	LBT *	LBT *	LBT *
105764	SBT *	SBT *	SBT *	SBT *	SBT *	SBT *	SBT *	SBT *
105765	MBT *	MBT *	MBT *	MBT *	MBT *	MBT *	MBT *	MBT *
105766	CBT *	CBT *	CBT *	CBT *	CBT *	CBT *	CBT *	CBT *
105767	SFB *	SFB *	SFB *	SFB *	SFB *	SFB *	SFB *	SFB *
105770	ISY *	ISY *	ISY *	ISY *	ISY *	ISY *	ISY *	ISY *
105771	DSY *	DSY *	DSY *	DSY *	DSY *	DSY *	DSY *	DSY *
105772	JPY *	JPY *	JPY *	JPY *	JPY *	JPY *	JPY *	JPY *
105773	SBS *	SBS *	SBS *	SBS *	SBS *	SBS *	SBS *	SBS *
105774	CBS *	CBS *	CBS *	CBS *	CBS *	CBS *	CBS *	CBS *
105775	TBS *	TBS *	TBS *	TBS *	TBS *	TBS *	TBS *	TBS *
105776	CMW *	CMW *	CMW *	CMW *	CMW *	CMW *	CMW *	CMW *
105777	MVW *	MVW *	MVW *	MVW *	MVW *	MVW *	MVW *	MVW *
1064NN	MIB	MIB	MIB	MIB	MIB	MIB	MIB	MIB
1065NN	LIB	LIB	LIB	LIB	LIB	LIB	LIB	LIB
11(0nn)NNN	AND •	AND •	AND •	AND •	AND •	AND •	AND •	AND •
11(1nn)NNN	JSB •	JSB •	JSB •	JSB •	JSB •	JSB •	JSB •	JSB •
12(1nn)NNN	JMP •	JMP •	JMP •	JMP •	JMP •	JMP •	JMP •	JMP •
13(0nn)NNN	IOR •	IOR •	IOR •	IOR •	IOR •	IOR •	IOR •	IOR •
13(1nn)NNN	ISZ •	ISZ •	ISZ •	ISZ •	ISZ •	ISZ •	ISZ •	ISZ •
14(0nn)NNN	ADA •	ADA •	ADA •	ADA •	ADA •	ADA •	ADA •	ADA •
14(1nn)NNN	ADB •	ADB •	ADB •	ADB •	ADB •	ADB •	ADB •	ADB •
15(0nn)NNN	CPA •	CPA •	CPA •	CPA •	CPA •	CPA •	CPA •	CPA •
15(1nn)NNN	CPB •	CPB •	CPB •	CPB •	CPB •	CPB •	CPB •	CPB •
16(0nn)NNN	LDA •	LDA •	LDA •	LDA •	LDA •	LDA •	LDA •	LDA •
16(1nn)NNN	LDB •	LDB •	LDB •	LDB •	LDB	LDB •	LDB	LDB •
17(0nn)NNN	STA •	STA •	STA •	STA •	STA •	STA •	STA •	STA •
17(1nn)NNN	STB •	STB •	STB •	STB •	STB •	STB •	STB •	STB •

Notes:
n Any binary numeric
N Any octal numeric
* May be used with or without a leading dot
• Indirect addressing

D

HP 1000 Computer Base and Extended Instruction Sets

Table D-1 is a listing of the base set of instructions (binary). Table D-2 presents a summary of the extended set of instructions for A-Series Computers. Table D-3 lists additional instructions for the A990 Computer.

Note The instruction set listed in each computer series reference manual takes precedence over these tables and should, therefore, be considered as the ultimate source for instruction sets.

Table D-1. Base Set of Instruction Codes

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		*LS	000	
				A/B	0	D/E	*RS	001			D/E	‡SL*	*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	0	L*E	101			0		L*E	101	
				A/B	0	0	S*E	110			0		S*E	110	
					000		NOP	000			000		NOP	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	
				.MPYD**†		100		000			000			000	
				.DIVD**†		100		001			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	← Number Of Bits →			
				RRL		000		001		0	0				
				LSRD†		001		000		1	1				
				LSLD†		000		000		1	1				
				ASRD†		100		000		1	1				
				ASLD†		100		000		1	0				

Notes: D/I, A/B, Z/C, D/E, H/C coded: 0/1.
* = A or B, according to bit 11.
** Second word is Memory Address.
† A990 only.

†CLE: Only this bit is required.
‡SL*: Only this bit and bit 11 (A/B as applicable) are required.

Table D-2. Summary of Extended Set of Instruction Codes for A-Series Computers (sheet 1 of 3)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	R	0	G	n	n	n	n	n	n	n	n	n

R	G	Opcode Block	Instruction Group
0	0	100NNN	Extended Arithmetic
0	1	101NNN	Extended Arithmetic, Code & Data Separation, and Index Register Insts.
1	0	104NNN	Extended Arithmetic
1	1	105NNN	Dynamic Mapping System, Virtual Memory Access, Scientific Instruction Set, Code & Data Separation , and Index Register Instructions.

100NNN Opcode Block

100000...100017	1	000	000	000	00n	nnn	–
100020...100037	1	000	000	000	01n	nnn	ASL
100040...100057	1	000	000	000	10n	nnn	LSL
100060...100077	1	000	000	000	11n	nnn	LSLD
100100...100117	1	000	000	000	01n	nnn	RRL
100120...100177							–
100200	1	000	000	010	000	000	MPY
100201...100377							–
100400	1	000	000	100	000	000	DIV
100401...100577							–
100600	1	000	000	110	000	000	JLA
100601...100777							–

101NNN Opcode Block

101000...101017	1	000	001	000	00n	nnn	–
101020...101037	1	000	001	000	01n	nnn	ASR
101040...101057	1	000	001	000	10n	nnn	LSR
101060...101077	1	000	001	000	11n	nnn	LSRD
101100...101117	1	000	001	001	00n	nnn	RRR
101120...101177							–
101200...101377	1	000	001	01n	nnn	nnn	–

N▶	0	1	2	3	4	5	6	7
10140N	–	–	–	–	–	–	.CCQA	.CACQ
10141N	.CZA	.CAZ	.CIQA	.ADQA	–	–	–	–
10142N	–	–	–	–	–	–	–	–
10143N	–	–	–	–	–	–	–	–
10144N	–	–	–	–	–	–	–	–
10145N	–	–	–	–	–	–	–	–
10146N	–	–	–	–	–	–	–	–
10147N	–	–	–	–	–	–	–	–

Notes: n Binary numeric
N Octal numeric

Table D-2. Summary of Extended Set of Instruction Codes for A-Series Computers (sheet 2 of 3)

101NNN Opcode Block (continued)

N▶	0	1	2	3	4	5	6	7
101500...101677	–	–	–	–	–	–	–	–
10170N	–	–	–	–	–	–	–	–
10171N	–	–	–	–	–	–	*	–
10172N	–	XLA2	XSA2	XCA2	XLA1	XSA1	XCA1	MB00
10173N	MB01	MB02	MB10	MB11	MB12	MB20	MB21	MB22
10174N	SAX	CAX	LAX	–	CXA	–	–	XAX
10175N	SAY	CAY	LAY	–	CYA	–	–	XAY
10176N	–	–	–	–	–	–	–	–
10177N	–	–	–	–	–	–	–	–

104NNN Opcode Block

104000	1	000	100	000	000	000	.MPYD
104001...104037							–
104040...104057	1	000	100	000	11n	nnn	ASLD
104060...104077	1	000	100	000	11n	nnn	ASRD
104100	1	000	100	001	000	000	.DIVD
104101...104177							–
104200	1	000	100	010	000	000	DLD
104201...104377							–
104400	1	000	100	100	000	000	DST
104401...104577							–
104600	1	000	100	110	000	000	JLB
104601...104777							–

105NNN Opcode Block

N▶	0	1	2	3	4	5	6	7
10500N	FAD	VADD	.TADD	VSUB	VMPY	VDIV	VSAD	VSSB
10501N	VSMY	VSDV	–	–	.DAD	–	–	–
10502N	FSB	DVADD	.TSUB	DVSUB	DVMPY	DVDIV	DVSAD	DVSSB
10503N	DVSMY	DVSDV	–	–	.DSB	–	–	–
10504N	FMP	–	.TMPY	–	–	–	–	–
10505N	–	–	–	–	.DMP	–	–	–
10506N	.FDV	–	.TDIV	–	–	–	–	–
10507N	–	–	–	–	.DDI	–	–	–
10510N	FIX	VPIV	.TFXS	VABS	.FIXD	VSUM	.TFXD	VNRM
10511N	VDOT	VMAX	VMAB	VMIN	.DSBR	VMIB	VMOV	VSWP
10512N	FLT	DVPIV	.TFTS	DVABS	.FLTD	DVSUM	.TFTD	DVNRM
10513N	DVDOT	DVMAX	DVMAB	DVMIN	.DDIR	DVMIB	DVMOV	DVSWP
10514N	–	–	–	–	–	–	–	–
10515N	–	–	–	–	–	–	–	–
10516N	–	–	–	–	–	–	–	–
10517N	–	–	–	–	–	–	–	–

Notes: n Binary numeric
N Octal numeric
* Place holder for .XLAB (future)

Table D-2. Summary of Extended Set of Instruction Codes for A-Series Computers (sheet 3 of 3)

105NNN Opcode Block (continued)								
N▶	0	1	2	3	4	5	6	7
10520N	–	–	–	.DNG	.DCO	.DFER	–	.BLE
10521N	.DIN	.DDE	.DIS	.DDS	.NGL	–	–	–
10522N	.XFER	–	.DSZ	.ENTR	.ENTP	.PWR2	.FLUN	.SETP
10523N	.PACK	.CFER	..FCM	..TCM	.ENTN	.ENTC	.CPM	.ZFER
10524N	.PMAP	–	–	–	.IRES	.JRES	–	.LBPC
10525N	.IMAP	–	.JMAP	–	.LPXR	.LPX	.LBPR	.LBP
10526N	–	–	–	–	–	–	–	–
10527N	–	–	–	–	–	–	–	–
10530N	.CPUID	.FWID	.WFI	.SIP	.WCS	.RCS	.WTM	.RTM
10531N	.WTC	.RTC	–	–	–	–	–	–
10532N	TAN	SQRT	ALOG	ATAN	COS	SIN	EXP	ALOGT
10533N	TANH	DPOLY	/CMRT	/ATLG	.FPWR	.TPWR	–	–
10534N	–	–	–	–	–	–	–	–
10535N	–	–	–	–	–	–	–	–
10536N	–	–	–	–	–	–	–	–
10537N	–	–	–	–	–	–	–	–
10540N	.PCALI	.PCALX	.PCALV	.PCALR	.PCALN.	.SDSP	.CCQB	.CBCQ
10541N	.CZB	.CBZ	.CIQB	.ADQB	–	.EXIT1	.EXIT2	.EXIT0
10542N	–	–	–	–	–	–	–	–
10543N	–	–	–	–	–	–	–	–
10544N	–	–	–	–	–	–	–	–
10545N	–	–	–	–	–	–	–	–
10546N	–	–	–	–	–	–	–	–
10547N	–	–	–	–	–	–	–	–
105500..105677	–	–	–	–	–	–	–	–
10570N	.LPMR	.SPMR	.LDMP	.STMP	.LWD1	.LWD2	.SWMP	.SIMP
10571N	.XJMP	.XJCQ	.LXMP	.SXMP	.LXMR	.SXMR	*	–
10572N	–	.XLB2	.XSB2	.XCB2	.XLB1	.XSB1	.XCB1	.MW00
10573N	.MW01	.MW02	.MW10	.MW11	.MW12	.MW20	.MW21	.MW22
10574N	SBX	CBX	LBX	STX	CXB	LDX	ADX	XBX
10575N	SBY	CBY	LBY	STY	CYB	LDY	ADY	XBY
10576N	ISX	DSX	JLY	LBT	SBT	MBT	CBT	SFB
10577N	ISY	DSY	JPY	SBS	CBS	TBS	CMW	MVW

Notes: n Binary numeric
N Octal numeric
* Place holder for .XLBB (future)

Table D-3. Additional Instruction Codes for A990 Computers

Opcode (octal)	Mnemonic	Description of Instruction
100060...100077	LSLD	LSL double integer format
101060...101077	LSRD	LSR double integer format
104000	.MPYD	Multiply, double integer format
104100	.DIVD	Divide, double integer format
104040...104057	ASLD	ASL, double integer format
104060...104077	ASRD	ASR, double integer format
105222	.DSZ	16-bit decrement and skip on zero
105247	.LBPC	LBD with byte pointers
105304	.WCS	Write control store
105305	.RCS	Read control store
105306	.WTM	Write timer register
105307	.RTM	Read timer register
105310	.WTC	Write clock chip register
105311	.RTC	Read clock chip register
105712	.LXMP	Load extended map set
105713	.SXMP	Store extended map set
105714	.LXMR	Load extended map register
105715	.SXMR	Store extended map register

E

Macro/1000 Assembler Operations

This appendix describes the Macro/1000 assembler (MACRO) operations. MACRO can be run interactively from an input device or through a command file to translate source programs into absolute or relocatable code. The absolute or relocatable form of the output is specified either in the runstring or in the macro control statement in the source program.

Macro Control Statement

The macro control statement must be the first statement in the source program, of the form:

`MACRO,` *p1*`,`*p2*`,`*p3*`,...,`*pn*

Commas are required to separate the order-independent options. Options specified in the control statement can be overridden by options specified in the runstring. Except when the 'M' option is given, the control statement may be continued on a continuation line if necessary.

A Absolute assembly. The addresses generated by the assembler are interpreted as absolute locations in memory. The program is a complete entity; no external symbols, common storage references, or entry points are included. Note that an absolute program cannot execute on RTE. (See note 1.)

R Relocatable assembly. The object program can be loaded anywhere in memory. All operands that refer to memory locations are adjusted as the program is loaded. External symbols, entry points and common storage references are allowed in the program. (See note 1.)

M Macro Library creation. Create a macro library and place in the destination file specified in the runstring. The destination file must be Type 1. (See note 2.)

L List output. Output an object code listing of the program to the list file or device. The listing includes both the opcode and the address of the operand if it is a memory reference instruction. (See notes 3 and 4.)

There are three suboptions with the L option:

=S Short listing. No macro expansion, no conditional assembly.

=M Medium listing. Expand macros, no conditional assembly.

=L Long listing. Expand macros, include conditional assembly.

When the suboptions are included, they must be given as L=S (no space between the option and the suboption characters).

Q List output. Output only the operand address for single-word memory reference instructions; that is, only the unrelocated address and not the opcode. The entire instruction will be listed otherwise. The suboptions =S, =M, and =L are also permitted with this option. (See notes 3 and 4.)

T Symbol Table. Include a listing of the symbol table in the output. When used with the M option, specifies inclusion of all macro library names in the list file.

C Cross-Reference Table. Include the cross-reference table in the listing. This table provides all references to statement labels, all external symbols and all user-defined opcodes, together with the source file line numbers at which they are defined and referenced.

I Generate Microcode Instructions. This option overrides the Macro/1000 code-generating feature that generates a JSB for all microcode instructions. For example, without this option, Macro/1000 would generate code for a "JSB .LBT" when it encounters the LBT opcode. The I option specifies that the microcode call for such instructions is to be generated. Refer to Chapter 3 for a description of the instructions this option affects. (In general the software is more flexible if the I option is not used, as the loader can then tailor the module to fit the hardware at load time.)

Note If this option is used, Macro/1000 will generate opcodes for the M/E/F-Series computers. These opcodes may or may not be identical to A-Series opcodes. See Appendix C and D for the differences in opcodes which exist between these computer series.

O Invoke the OLDRE utility. This option converts the format of the relocatable output file of Macro/1000 to be compatible with earlier RTE systems and generators. The OLDRE utility must be loaded on your system for this option to work. Refer to the *Utility Programs Reference Manua*l for more information on OLDRE.

S Enables symbolic debug mode. Must be loaded by LINK. Refer to the *Symbolic Debug/1000 User's Manual*, part number 92860-90001, for details on the debugger.

N,Z Selective Assembly. These options are included for backward compatibility with ASMB. A description of their use is contained in Appendix J.

D Ignore; this option is for future reference.

P,B, F,X Ignore; these options are included for backward compatibility with ASMB.

+DC=<851002
Software date code to be installed in word 24 of the NAM records of all modules processed under this control statement. The date is of the form <YYMMDD, where < is optional, YY is the last two digits of the calendar year, MM is the month and DD is the day of the month. The date is converted to the number of days since January 1, 1970 and put in the NAM record. This code is intended to allow high level compilers (such as Pascal) to put their revision level in the relocatable. This parameter cannot be overridden in the runstring.

+SF=filedescriptor

Normally MACRO puts the source file descriptor in the NAM record. This control causes MACRO to put 'file descriptor' in the NAM record instead. This code is intended to allow high level compilers (such as Pascal) to put the actual source file descriptor in the NAM record. The file descriptor is used by such routines as DEBUG. This parameter cannot be overridden in the runstring.

Note

1. Options A and R are mutually exclusive. Both may not appear in the control statement. If neither A nor R is specified, R is assumed.
2. Only the L,Q, and T options can be used with the M option.
3. If both L and Q are specified, the last one specified is used.
4. If no suboptions are used with the L or Q options, =S is assumed.

Runstring Parameters

Call MACRO with the following runstring, using commas as placeholders if you omit a parameter in the runstring:

```
[RU,]MACRO,source[,list[,dest ┌,&.RS1=string1         ┐┐┐
                              │,&.RS2=string2         │││
                              │,WORK=filedescriptor   │││
                              │,OPT=opt               │││
                              │,LINES=pg              │││
                              │,opt                   │││
                              └,pg                    ┘┘┘
```

source The source file to be assembled is the only required parameter. Source may be a file, an interactive device LU, or the LU of a non-interactive device such as a magnetic tape or a cartridge tape unit. If source is a file and it is not found and has no type extension, MACRO appends .MAC and tries again.

list The file name or device LU to which the listing is directed. If the file does not exist, it is created. If this parameter is omitted, no listing is produced. Enter a dash (−) or an at sign (@) in this position to default the list parameter. The defaults are described in the section on Default Output File Formats.

If this parameter is defaulted, the source file name must be of the form &NAME (& as the first character) or NAME.MAC (.MAC type extension). File names of any other form are rejected by MACRO, aborting the assembly.

dest The file name or device LU to which the binary output is directed. If the file does not exist, it is created. The dest parameter may be defaulted by entering a dash (−) or an at sign (@) in this position. Refer to the section Default Output File Formats for more information.

If this parameter is defaulted, the source file name must be of the form &NAME (& as the first character) or NAME.MAC (.MAC type extension). File names of any other form are rejected by MACRO, aborting the assembly.

The following parameters may appear in any order:

pg
`LINES=`*pg*
Is a decimal number specifying the number of lines per page for the list device. If this parameter is omitted, the page size defaults to 55 lines.

opt
`OPT=`*opt*
Can be any of the options (except the ignored options) that are allowed in the control statement, and will override those options. In the runstring, the options must be specified without intervening commas (as RLTC).

An override option, P, also is allowed in the runstring. This option, which must be the only one specified in the string, directs that only the object code and error messages will be output, with no further action by MACRO.

`&.RS1=`
`&.RS2=`
Assign character values to global system assembly-time variables of the same names in the source program. If the characters include a blank or a comma, the string must be surrounded with single quotes.

`WORK=`
Specifies a filedescriptor that will be used to build the scratch file-descriptor. The scratch files are created on the specified directory and are of the specified size. If this parameter is omitted, 96-block files are created on the system directory /SCRATCH. Refer to the Default Output File Formats section for more information.

Default Output File Formats

When the list or destination filedescriptors are defaulted, MACRO constructs file names as follows:

Source Filedescriptor	List File	Destination File
&*name*	'*name*	$*name* (Macro Library)
		%*name* (Relocatable code)
		!*name* (Absolute code)
name.MAC	*name*.LST	*name*.MLB (Macro Library)
		name.REL (Relocatable code)
		name.ABS (Absolute code)

File attributes can be specified by expanding the default characters @ or − to include the desired attributes. For example:

```
/A3RELS/-::::40
```

specifies the directory pathname /A3RELS, defaults the output file name to that of the source under the above transform rules, and specifies the file size as 40 blocks. Remember that when the name is defaulted, the source file name must be of the form &*name* or *name*.MAC. Any other file name format is rejected by
MACRO.

If not provided, file attributes other than type and size default to those of the source filedescriptor. Size defaults to 48 blocks. The type field is always forced to:

Type	
3	List file
1	Macro Library file
5	Relocatable code file
7	Absolute code file

When the source filedescriptor is of the form *name*.MAC, you can both default the list and/or destination file names and suppress the addition of the type extension by entering the default as:

```
RU,MACRO,SOURCE.MAC,-.,-
```

Placing the period after the dash tells MACRO to construct a filedescriptor with the same attributes as the source filedescriptor, but with a null type extension. In the above example, the list file will be named SOURCE and the destination filedescriptor will default to SOURCE.MLB, SOURCE.REL, or SOURCE.ABS as appropriate.

MACRO builds a type 3 work file (IF) to hold the source file and flags, a type 1 file (SW) to hold tables and, if the destination is an absolute code file, a type 5 file (AB) to hold the intermediate results of the translation. When the work file (WORK=) option is omitted, they are created as 96-block files on the system directory /SCRATCH. The WORK= option allows you to specify a different directory and/or file size.

Examples

1. `RU,MACRO,&PROGA,-,-`

   This example schedules MACRO to assemble the source code in file &PROGA. The list file name defaults to 'PROGA, and the destination file defaults to %PROGA. The type of the output (relocatable or absolute) is as defined in the source program MACRO control statement.

2. `RU,MACRO,&FILE,-,-,,,&.RS1=1,&.RS2=2`

   The source file contains the code:

   ```
   MACRO,L,T,R
         NAM FILE
         AIF &.RS1=1
            :
            :               *assemble this section of code
            :
         AELSEIF &.RS2=2
            :
            :               *assemble this section of code
            :
         AENDIF
       END  FILE
   ```

   The defaults are as in example 1. The &.RS1 and &.RS2 flags are set and will result in assembling the flagged areas of code.

3. `MACRO,&FIL2,-,-,,,&.RS1=&FILE3`

   The source file contains the code:

   ```
   MACRO,L,T,R
        NAM FILE2
        INCLUDE &.RS1      *assemble &FILE3
           :
           :
           :
     END  FILE
   ```

   In this example, the &.RS1 flag identifies &FILE3. This file will be assembled when the source is assembled by MACRO.

4. `MACRO,&fil1:sc:50,‘list::55,-,,a`

   In this example, the list file is specified as 'LIST and will be placed on disk volume 55. The destination file will default to !FIL1:sc:50. The A option specifies an absolute code file.

5. `MACRO abcd.mac`

   This example schedules MACRO to assemble source file ABCD.MAC, defaulting the list file to the interactive input device (generally your terminal), and defaulting to 55 lines per page. No destination file is generated. Note that the runstring can be entered in either uppercase or lowercase letters.

6. `RU MACRO ABCD.MAC,@.,@,28,ACL=L`

   This example schedules MACRO to assemble source file ABCD.MAC, directing the output to list file ABCD with no type extension (period entered following the defaulting "@" character), and the destination to absolute code file ABCD.ABS (default extension, A option given). The number of lines per page is specified as 28. The L=L option specifies a long listing, with macros and the conditional assembly expanded. The C option specifies inclusion of the cross-reference table in the list file.

7. `RU,MACRO, ABCD.MAC,-,EFGH,,TQC`

   This example schedules MACRO to assemble source file ABCD.MAC, defaulting the list file to ABCD.LST, and specifying the destination file as EFGH. The option T directs that a symbol table is to be listed in the list file; option Q directs that the memory reference instructions in the object code listing are to appear as addresses only (no opcode listing); option C directs that a cross-reference table is to be listed in the list file.

8. `MACRO,&INT4,/LISTDIR/@::::200,/RELDSC/@,,,WORK=/BIGSCR/::::2000`

   This example schedules MACRO to assemble source file &INT4, defaulting the list file name but specifying the directory /LISTDIR and the file size of 200 blocks. The destination filedescriptor is defaulted, but the directory /RELDSC is specified. The work files, each 2000 blocks in size, will be created in directory /BIGSCR. These files will be named IF*nnnn*, to hold the source file and flags, and SW*nnnn*, to hold tables. The *nnnn* is replaced with a unique name generated by MACRO to specifically identify the files.

Messages During Assembly

Macro/1000 searches for the source file under the filedescriptor given in the runstring. If the file does not exist, Macro/1000 issues the message:

```
Macro: No such file     File = <filedescriptor>
```

If the type extension in the filedescriptor is blank, Macro/1000 first searches under the filedescriptor, and then searches under the filedescriptor with a .MAC type extension. If the source file does not exist, the following message is issued:

```
Macro: No such file     File = <file>.MAC
```

The .MAC type extension will always appear in the message in this case.

If the macro message catalog file does not exist, Macro/1000 attempts to create it on the /SYSTEM directory or on the default FMGR disk, if there is no system directory. If this directory is write-protected to the user, this will fail with the appropriate error message. In this case, the system manager should install the message catalog file as specified in the next section, “Installing Macro/1000.”

If an error is found in the assembler control statement or the runstring, Macro/1000 aborts the assembly after issuing the following message to your interactive input device:

```
Illegal option in runstring or control statement
```

If a system error occurs during the assembly, Macro/1000 issues the appropriate error message, identifies the file being processed when the error occurs, and aborts.

At the end of the assembly, Macro/1000 issues the message:

```
Macro/1000  Rev.5000  870429 : No errors found
```

or issues the following error status messages:

```
 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>
```

Following these messages, Macro/1000 prints an error description (see Appendix A for format) for each unique error number (*eee*) appearing in the status messages.

```
    Macro/1000 Rev.5000 870429 : xx errors found
```

where:

eee is the error number.

nn is the line number at which the error was detected.

mmm is the line number of the macro definition at which the error occurred. This phrase is included only if an error occurs inside the macro definition.

iii is the file number of an included file. This phrase is included only if an error occurs inside an include file. In this case, the nn represents the include file line number.

xx is the total error count.

Macro/1000 returns the number of errors to the program that scheduled the assembler as the first return parameter. This parameter can be retrieved through a call to the library subroutine RMPAR. For CI users, this is RETURN1. For FMGR users, this is 1P.

Installing Macro/1000

Macro/1000 can be installed online using the transfer file #MACRO as:

```
RU,LINK,#MACRO
```

When first run, Macro/1000 creates its message catalog file. To do this, it must be run with the capability to create a file on the /SYSTEM directory or the default FMGR cartridge if there is no system directory. To just create the file, you may run Macro/1000 as follows:

```
RU, MACRO, –1
```

This forces the message catalog file to be created or recreated. To list the message catalog file, run Macro/1000 as follows:

`RU,MACRO,–2,`*listfile*

The message catalog file is named MACRO*rev*.ERR where *rev* is the current revision code (for example, MACRO5000.ERR). When you install a new revision of Macro/1000, you may want to purge the old message catalog file.

If there is no /SYSTEM directory, the message catalog file name is M.E*mn*, where *m* is the major revision number (5 in the example) and *n* is the minor rev number. The minor rev number is the third digit of the four-digit revision code (0 in the example).

Using Old Macro Libraries

Macro/1000, as of revision 6.0, verifies that macro libraries were processed with a MACRO which has the same opcodes as the using MACRO. This is important since redefined opcodes must be in the same place in the opcode table. Error 21 results if this verification fails.

The −3 option allows you to recompile old macro libraries. This option may also be used to get a listing of a macro library. The command, using the −3 option, is:

`RU,MACRO,–3,`*listfile*`,`*maclibfile*

The file "MACLIBFILE" will be reprocessed in place and will have the correct duplicate opcode flags for use with the Macro/1000 version doing the reprocessing. Note that the opcode revision used to verify a macro library may vary from or change with the Macro/1000 revision.

The file "MACLB is supplied for use with Macro/1000. This file is the source of the macros described in Appendix K. To install this file, run Macro/1000 against it with the command:

`RU,MACRO,"MACLB,`*list*`,`*mlib*

where:

list is the list filedescriptor or device to which you want the listing to be directed, or 0 if no listing is desired.

mlib is the filedescriptor for the macro library. A recommended descriptor is:

```
MACLIB.MLB::LIBRARIES
```

To use the MACLIB.MLB file (or any other macro library), your source code must include the line:

```
MACLIB filedescriptor
```

In searching for the file in the CI file-system environment, MACRO will default to user-defined search path 3 (refer to the UDSP description in the CI User's Manual). If this fails, MACRO then searches the same directory that contains the source file.

If your system supports VC+, you also should have the file ”CDSLB. Install this file in the directory LIBRARIES as described above for ”MACLB, with the recommended file name CDSLB.MLB.

You should also install the CDSONOFF macro library in the directory, LIBRARIES. The recommended name is $CDSONOFF.MLB. $CDSONOFF.MLB does an include operation on ”MACLB so this library must also be available to MACRO when building the CDSONOFF library.

Note that if the utility OLDREC is required for your system, it should also be loaded.

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. A sample cross-reference listing is given on the following page.

The table entry format is:

symbol `.....` *sd*`[(`*i*`)]:`   *sr1*`[*]`   *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.

sr_i[*] — 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, and are listed in the Cross-Reference Table with SD=0.

If the symbol is not referenced in the file, the message:

```
symbol not referenced
```

is printed after the defining statement number. If a symbol is defined more than once, all definitions will appear in the cross-reference.

Note The symbols *** RELOC **
*** ORG ****
*** ORB ****
*** ORR ****

will appear in the cross-reference table output if statements by the same name appear in the source.

```
PAGE#  1     OPERM.MAC::MAC                  12:14 PM  THU., 28  MAY , 1987

00001                macro,q,r,c
00002                         nam operm
00003                         maclib $maclb
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 000000R          lda init      ; init is referenced here
00011  00005 001300           rar
00012  00006 000001R          sta change    ; change is altered here
00013  00007 000014R          ldb =d6       ; use a literal
00014  00010 000011R          jmp over      ; over is a volatile reference

00015                ;
00016                ;
00017  00011 000000  over     nop           ; over is defined here
00018  00012                  call ’exec’ =d6 ; call macro to exit
       00014 000006
00019                         end operm

PAGE#  2     OPERM.MAC::MAC                  12:14 PM  THU., 28  MAY , 1987

* – Volatile reference (store, jmp, call...)

  *** RELOC **  . . . .7(1):     8
 A  . . . . . . . . . . .0:  Symbol not referenced
 ABC  . . . . . . . . . .8(1):  Symbol not referenced
 B  . . . . . . . . . . .0:  Symbol not referenced
 CHANGE . . . . . . . . .5(1):    12*
 EXEC . . . . . . . . . 18:    18*
 INIT . . . . . . . . . .4(1):    10
 NOT.USED . . . . . . . .6(1):  Symbol not referenced
 OPERM  . . . . . . . . .9:    19*
 OVER . . . . . . . . . 17:    14*

Macro/1000 Rev.5000 870429 :  No errors found
```

G

HP Character Set

Effect of Control Key *

					000-037B		040-077B		100-137B		140-177B	
7 6 5					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits				Col.	0	1	2	3	4	5	6	7
4	3	2	1	Row								
0	0	0	0	0	NUL	DLE	SP	0	@	P	‘	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	”	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	’	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	–	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

32 Control Codes

Upshifted Lowercase

64 Character Set

96 Character Set

128 Character Set

Example: The representation for the character “K” (column 4, row 11) is

Bit	7	6 5 4	3 2 1
Binary	1	$\underbrace{0\ 0\ 1}$	$\underbrace{0\ 1\ 1}$
Octal	1	1	3

Note: * Depressing the Control Key while typing an uppercase letter produces the corresponding control code on most terminals. For example, Control-H is a backspace.

Table G-1. Hewlett-Packard Character Set for Computer Systems

This table shows Hewlett-Packard'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		Mnemonic	Graphic[1]	Meaning
	Left Byte	Right Byte			
0	000000	000000	NUL	N_U	Null
1	000400	000001	SOH	S_H	Start of Heading
2	001000	000002	STX	S_X	Start of Text
3	001400	000003	EXT	E_X	End of Text
4	002000	000004	EOT	E_T	End of Transmission
5	002400	000005	ENQ	E_Q	Enquiry
6	003000	000006	ACK	A_K	Acknowledge
7	003400	000007	BEL	🔔	Bell, Attention Signal
8	004000	000010	BS	B_S	Backspace
9	004400	000011	HT	H_T	Horizontal Tabulation
10	005000	000012	LF	L_F	Line Feed
11	005400	000013	VT	V_T	Vertical Tabulation
12	006000	000014	FF	F_F	Form Feed
13	006400	000015	CR	C_R	Carriage Return
14	007000	000016	SO	S_O	Shift Out } Alternate
15	007400	000017	SI	S_I	Shift In } Character Set
16	010000	000020	DLE	D_L	Data Link Escape
17	010400	000021	DC1	D_1	Device Control 1 (X-ON)
18	011000	000022	DC2	D_2	Device Control 2 (TAPE)
19	011400	000023	DC3	D_3	Device Control 3 (X-OFF)
20	012000	000024	DC4	D_4	Device Control 4 ($\overline{\text{TAPE}}$)
21	012400	000025	NAK	N_K	Negative Acknowledge
22	013000	000026	SYN	S_Y	Synchronous Idle
23	013400	000027	ETB	E_B	End of Transmission Block
24	014000	000030	CAN	C_N	Cancel
25	014400	000031	EM	E_M	End of Medium
26	015000	000032	SUB	S_B	Substitute
27	015400	000033	ESC	E_C	Escape[2]
28	016000	000034	FS	F_S	File Separator
29	016400	000035	GS	G_S	Group Separator
30	017000	000036	RS	R_S	Record Separator
31	017400	000037	US	U_S	Unit Separator
127	077400	000177	DEL	■	Delete. Rubout[3]

Table G-1. Hewlett-Packard Character Set for Computer Systems (continued)

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

Table G-1. Hewlett-Packard Character Set for Computer Systems (continued)

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
64	040000	000100	@	Commercial At
65	040400	000101	A	Upper Case Letters
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	[	Left (opening) Bracket
92	056000	000134	\	Backslash. Reverse Slant
93	056400	000135	]	Right (closing) Bracket
94	057000	000136	^ ↑	Caret. Circumflex: Up Arrow[4]
95	057400	000137	_ ←	Underline: Back Arrow[4]

Table G-1. Hewlett-Packard Character Set for Computer Systems (continued)

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
96	060000	000140	‘	Grave Accent[5]
97	060400	000141	a	Lower Case Letters[5]
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	{	Left (opening) Brace[5]
124	076000	000174	|	Vertical Line[5]
125	076400	000175	}	Right (closing) Brace[5]
126	077000	000176	~	Tilde, Overline[5]

Note 1: 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.

Note 2: Escape is the first character of a special control sequence. For example, ESC followed by “J” clears the display on an HP 2640 terminal.

Note 3: Delete may be displayed as “_”, “@”, or space.

Note 4: Normally, the caret and underline are displayed. Some devices substitute the up arrow and the back arrow.

Note 5: Some devices upshift lowercase letters and symbols (‘ through ~) to the corresponding uppercase character (@ through ^). For example, the left brace would be converted to a left bracket.

Table G-2. 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
&	†[1]	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

Note 1: †The ASCII code 046 is converted to the BCD code for a space (20) when writing data onto a 7-track tape.

H

Relocatable Record Formats

This appendix identifies the format of the various relocatable records. Macro/1000 generates only those records identified by a “7” in bits 13-15 of the record ident (word 2) except for the lindex records (generated by LINDX) and the indxr records (generated by INDXR). These are called extended relocatable records.

OLDRE converts these records, if possible, to records that have record idents other than 7 in bits 13−15. All loaders recognize OLDRE type records, but all loaders do not recognize extended relocatable records.

NAM

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8">&lt;Zero&gt;</td><td></td></tr>
<tr><td>2</td><td colspan="3">1</td><td colspan="13">&lt;Zero&gt;</td><td>Record ident</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td></td></tr>
<tr><td>4</td><td colspan="16"></td><td></td></tr>
<tr><td>5</td><td colspan="16">Symbol</td><td></td></tr>
<tr><td>6</td><td colspan="8"></td><td colspan="8">&lt;Blank&gt;</td><td></td></tr>
<tr><td>7</td><td>KN</td><td colspan="15">Program size</td><td>KN:
0 - Size known
1 - Size uknown</td></tr>
<tr><td>8</td><td colspan="16">Base page size</td><td></td></tr>
<tr><td>9</td><td colspan="16">Common size</td><td></td></tr>
<tr><td>10</td><td colspan="16">Program type</td><td></td></tr>
<tr><td>11</td><td colspan="16">Priority</td><td></td></tr>
<tr><td>12</td><td colspan="16">Resolution code</td><td></td></tr>
<tr><td>13</td><td colspan="16">Execution multiple</td><td></td></tr>
<tr><td>14</td><td colspan="16">Hours</td><td></td></tr>
<tr><td>15</td><td colspan="16">Minutes</td><td></td></tr>
<tr><td>16</td><td colspan="16">Seconds</td><td></td></tr>
<tr><td>17</td><td colspan="16">10s of milliseconds</td><td></td></tr>
<tr><td>18</td><td colspan="16">Comment (variable length)</td><td></td></tr>
</table>

Figure H-1. NAM Record

XNAM

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8"><Zero></td><td></td></tr>
<tr><td>2</td><td colspan="3">7</td><td colspan="7">Subtype = 1</td><td colspan="6">Offset of size word</td><td>Record ident</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td></td></tr>
<tr><td>4
5</td><td colspan="16">Local EMA size</td><td></td></tr>
<tr><td>6</td><td colspan="16">Save size</td><td></td></tr>
<tr><td>7</td><td>KN</td><td colspan="15">Program size</td><td>KN:
0 - Size known
1 - Size unknown</td></tr>
<tr><td>8</td><td colspan="16">Base page size</td><td></td></tr>
<tr><td>9</td><td colspan="16">Common size</td><td></td></tr>
<tr><td>10</td><td colspan="16">Program type</td><td></td></tr>
<tr><td>11</td><td colspan="16">Priority</td><td></td></tr>
<tr><td>12</td><td colspan="16">Resolution code</td><td></td></tr>
<tr><td>13</td><td colspan="16">Execution multiple</td><td></td></tr>
<tr><td>14</td><td colspan="16">Hours</td><td></td></tr>
<tr><td>15</td><td colspan="16">Minutes</td><td></td></tr>
<tr><td>16</td><td colspan="16">Seconds</td><td></td></tr>
<tr><td>17</td><td colspan="16">10’s of milliseconds</td><td></td></tr>
<tr><td>18</td><td>C</td><td colspan="15">Pure code size</td><td>C:
0 - Not CDS
1 - CDS module</td></tr>
<tr><td>19</td><td colspan="16">Year of compilation</td><td></td></tr>
<tr><td>20</td><td colspan="9">Julian day</td><td colspan="7">Hour</td><td></td></tr>
<tr><td>21</td><td colspan="8">Minutes</td><td colspan="8">Seconds</td><td></td></tr>
<tr><td>22</td><td colspan="16">Literal count</td><td></td></tr>
</table>

Figure H-2. Extended NAM Record (XNAM)

XNAM (continued)

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td></td><td colspan="16"></td></tr>
<tr><td>23</td><td colspan="16">Revision of producer (days since 1970)</td></tr>
<tr><td>24</td><td colspan="16">Revision of preprocessor or −1 (days since 1970)</td></tr>
<tr><td>25</td><td colspan="8">Words of comment</td><td colspan="8">Words in symbol</td></tr>
<tr><td>26</td><td colspan="16">Symbol (variable length)</td></tr>
<tr><td></td><td colspan="16">Comment (variable length)</td></tr>
<tr><td></td><td colspan="16">Length of file name (−bytes)</td></tr>
<tr><td></td><td colspan="16">Source file name (variable length)</td></tr>
</table>

Figure H-2. Extended NAM Record (XNAM) (continued)

ENT

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8">&lt;Zero&gt;</td><td></td></tr>
<tr><td>2</td><td colspan="3">2</td><td colspan="9">&lt;Zero&gt;</td><td colspan="4">Entry count</td><td>Record ident</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td></td></tr>
<tr><td>4</td><td colspan="16" rowspan="2">First symbol</td><td></td></tr>
<tr><td>5</td><td></td></tr>
<tr><td>6</td><td colspan="8"></td><td colspan="5">&lt;Zero&gt;</td><td colspan="3">R</td><td></td></tr>
<tr><td>7</td><td colspan="16">Unrelocated address (or replacement value)</td><td></td></tr>
<tr><td></td><td colspan="16">Entry count - 1 repeats of above 4 word group</td><td></td></tr>
</table>

Figure H-3. ENT Record

XENT

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Record length								<Zero>							
2	7			Subtype = 2							Entry count					
3	Checksum															
4	E	Symbol ID number											MR			
5	Unrelocated value or offset															
6																
7	Words in info block								Words in symbol							
8	Symbol (variable length)															
	Match cond		info block (variable length) entry point characteristics like parameter count/type and version number													
	entry count -1 additional packets each 4 + words in info block + words in symbol words long															

Record ident

E: EMA (MR=5)

MR:
0 – absolute
1 – program
2 – base page
3 – common
4 – pure code
5 – local EMA
6 – SAVE area

Figure H-4. Extended ENT Record (XENT)

EXT

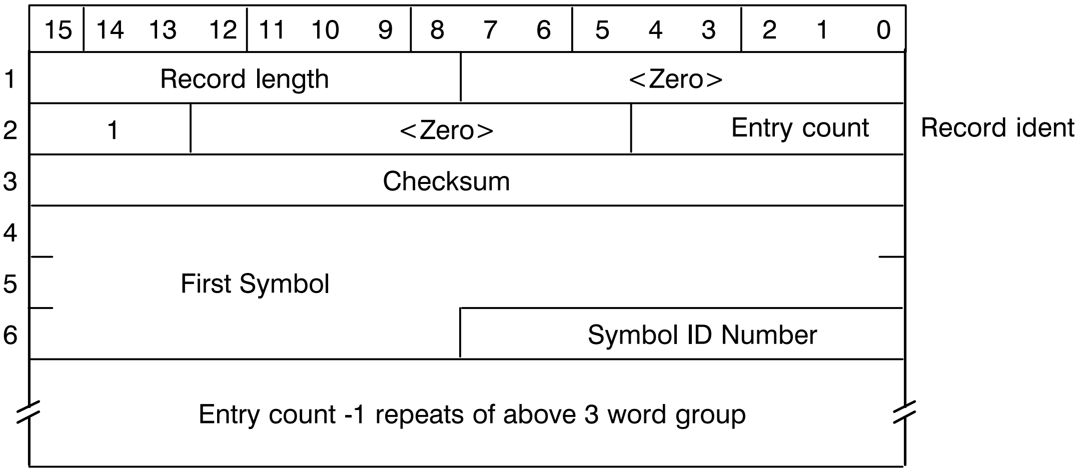


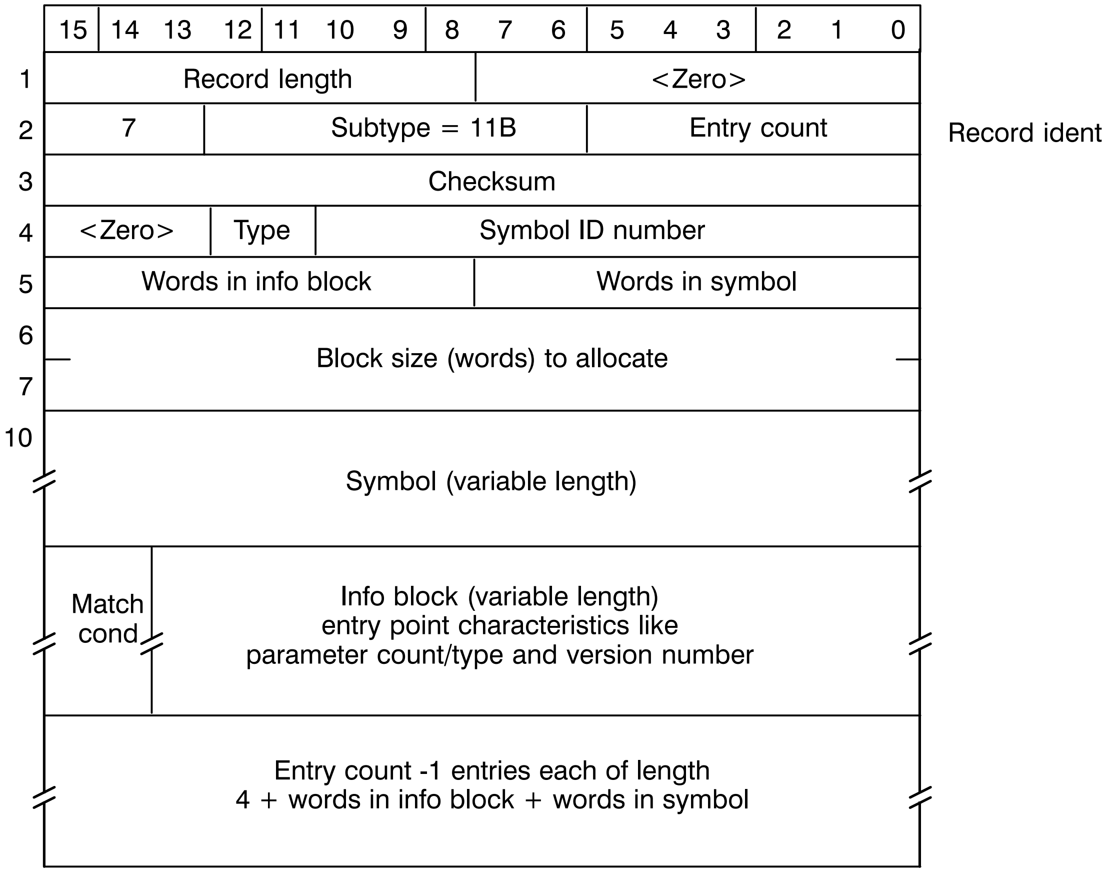
Figure H-5. EXT Record

XEXT

<table>
<tr><td></td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8">&lt;Zero&gt;</td><td></td></tr>
<tr><td>2</td><td colspan="3">7</td><td colspan="7">Subtype = 4</td><td colspan="6">Entry count</td><td>Record ident</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td></td></tr>
<tr><td>4</td><td>W</td><td colspan="4">&lt;Zero&gt;</td><td colspan="11">Symbol ID number</td><td>W : weak external</td></tr>
<tr><td>5</td><td colspan="8">Words in info block</td><td colspan="8">Words in symbol</td><td></td></tr>
<tr><td>6</td><td colspan="16">Symbol (variable length)</td><td></td></tr>
<tr><td></td><td colspan="2">Match cond</td><td colspan="14">Info block (variable length)
entry point characteristics like
parameter count/type and version number</td><td></td></tr>
<tr><td></td><td colspan="16">Entry count -1 repeats each of length
2 + words in info block + words in symbol</td><td></td></tr>
</table>

Figure H-6. Extended EXT Record (XEXT)

ALLOCATE



<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8">&lt;Zero&gt;</td><td></td></tr>
<tr><td>2</td><td colspan="3">7</td><td colspan="7">Subtype = 11B</td><td colspan="6">Entry count</td><td>Record ident</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td></td></tr>
<tr><td>4</td><td colspan="3">&lt;Zero&gt;</td><td colspan="2">Type</td><td colspan="11">Symbol ID number</td><td></td></tr>
<tr><td>5</td><td colspan="8">Words in info block</td><td colspan="8">Words in symbol</td><td></td></tr>
<tr><td>6</td><td colspan="16" rowspan="2">Block size (words) to allocate</td><td></td></tr>
<tr><td>7</td><td></td></tr>
<tr><td>10</td><td colspan="16">Symbol (variable length)</td><td></td></tr>
<tr><td></td><td colspan="2">Match cond</td><td colspan="14">Info block (variable length)
entry point characteristics like
parameter count/type and version number</td><td></td></tr>
<tr><td></td><td colspan="16">Entry count -1 entries each of length
4 + words in info block + words in symbol</td><td></td></tr>
</table>

type = 0	Named COMMON (program allocate)
type = 1	Named SAVE COMMON (SAVE allocate)
type = 2	Named EMA COMMON (EMA allocate)
type = 3	Reserved

Figure H-7. ALLOCATE Record

EMA

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Record length								<Zero>							
2	6			<Zero>			EMA size (in pages)									
3	Checksum															
4	Symbol															
5																
6									Symbol ID number							
7	<Zero>											MSEG size				

Figure H-8. EMA Record

MSEG

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Record length								<Zero>							
2	7			Subtype = 10B							MSEG size					
3	Checksum															

Record ident

MSEG size: 1-29

Figure H-9. MSEG Record

DBL

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8">&lt;Zero&gt;</td><td></td></tr>
<tr><td>2</td><td colspan="3">3</td><td colspan="5">&lt;Zero&gt;</td><td colspan="2">Z/C</td><td colspan="6">Word count</td><td></td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td></td></tr>
<tr><td>4</td><td colspan="16">Unrelocated load address</td><td></td></tr>
<tr><td>5</td><td colspan="3">1st R</td><td colspan="3">2nd R</td><td colspan="3">3rd R</td><td colspan="3">4th R</td><td colspan="3">5th R</td><td>0</td><td></td></tr>
<tr><td></td><td colspan="16">16-bit absolute value</td><td>R = 0</td></tr>
<tr><td></td><td>I</td><td colspan="15">15-bit program relocatable address</td><td>R = 1</td></tr>
<tr><td></td><td>I</td><td colspan="15">15-bit base page relocatable address</td><td>R = 2</td></tr>
<tr><td></td><td>I</td><td colspan="15">15-bit common relocatable address</td><td>R = 3</td></tr>
<tr><td></td><td>I</td><td colspan="4">Opcode</td><td colspan="3">&lt;Zero&gt;</td><td colspan="8">Symbol ID number</td><td>R = 4</td></tr>
<tr><td></td><td>I</td><td colspan="4">Opcode</td><td colspan="3">&lt;Zero&gt;</td><td colspan="6">Symbol ID number</td><td colspan="2">MR</td><td rowspan="2">R = 5</td></tr>
<tr><td></td><td colspan="16">Unrelocated value or offset</td></tr>
<tr><td></td><td colspan="3">6th R</td><td colspan="3">7th R</td><td colspan="3">8th R</td><td colspan="3">9th R</td><td colspan="3">10th R</td><td>0</td><td></td></tr>
<tr><td></td><td colspan="14">&lt;Zero&gt;</td><td colspan="2">MR</td><td rowspan="2">R = 6</td></tr>
<tr><td></td><td colspan="15">Relocatable address</td><td>LR</td></tr>
</table>

Z/C:
0 = basepage
1 = program
2 = absolute
3 = common

MR:
0 = program
1 = base page
2 = common
3 = absolute

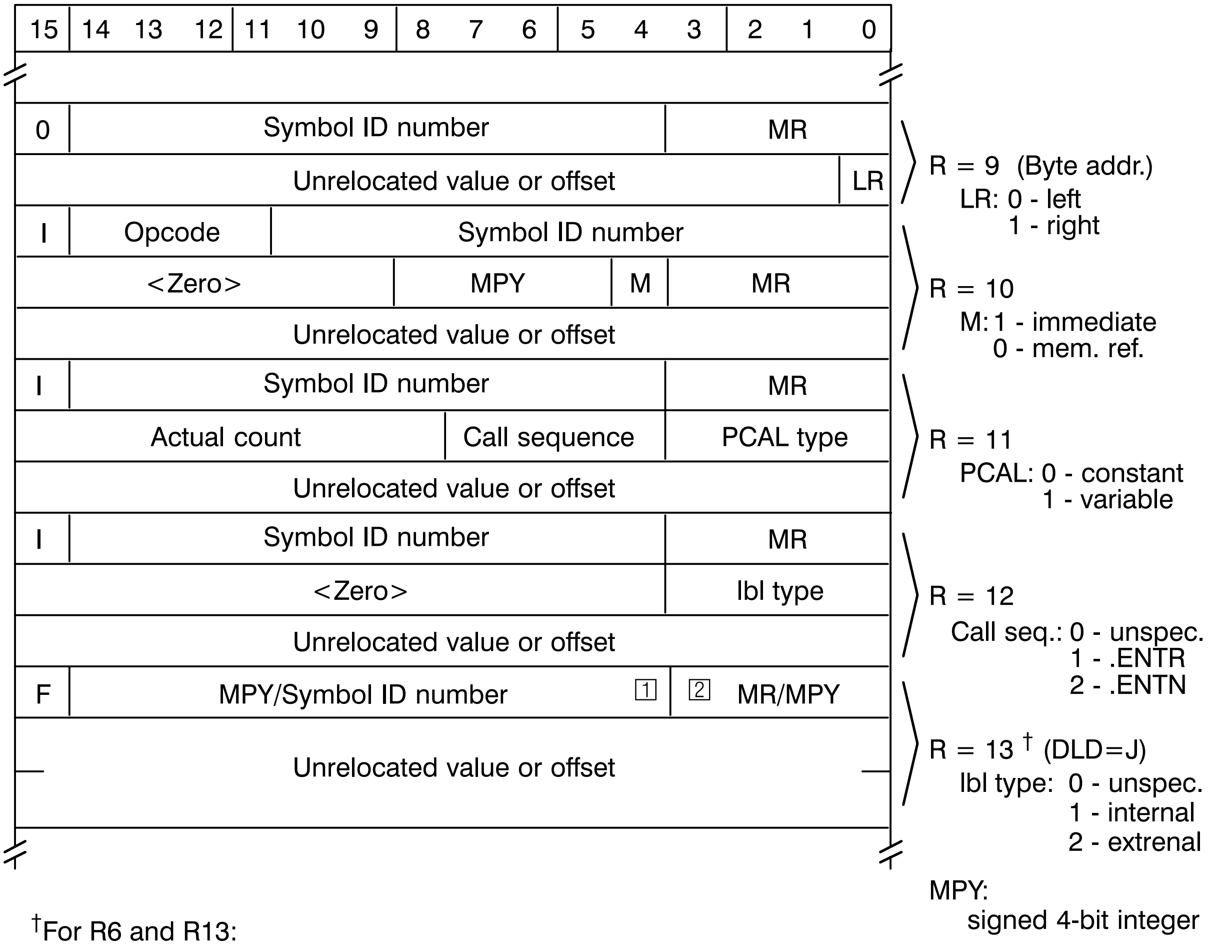
Figure H-10. DBL Record

XDBL

<table>
<tr><td></td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>B:</td></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8">&lt;Zero&gt;</td><td>0 – no break
1 – break allowed</td></tr>
<tr><td>2</td><td colspan="3">7</td><td colspan="7">Subtype = 3</td><td colspan="5">&lt;Zero&gt;</td><td>B</td><td>Record ident</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td rowspan="5">MR:
0 – absolute
1 – program
2 – base page
3 – common
4 – pure code
5 – local EMA
6 – SAVE area</td></tr>
<tr><td>4</td><td>0</td><td colspan="11">Symbol ID number</td><td colspan="4">MR</td></tr>
<tr><td>5</td><td colspan="16" rowspan="2">Unrelocated load address</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td><td colspan="8">Words to relocate</td><td colspan="8">No. of “R” values</td></tr>
<tr><td></td><td colspan="4">1st R</td><td colspan="4">2nd R</td><td colspan="4">3rd R</td><td colspan="4">4th R</td><td></td></tr>
<tr><td></td><td colspan="16">Absolute value</td><td>R = 0</td></tr>
<tr><td></td><td colspan="16">Program relocatable value</td><td>R = 1</td></tr>
<tr><td></td><td colspan="16">Base page relocatable value</td><td>R = 2</td></tr>
<tr><td></td><td colspan="16">Common relocatable value</td><td>R = 3</td></tr>
<tr><td></td><td colspan="4">5th R</td><td colspan="4">6th R</td><td colspan="4">7th R</td><td colspan="4">8th R</td><td></td></tr>
<tr><td></td><td>I</td><td colspan="4">Opcode</td><td colspan="11">Symbol ID number</td><td>R = 4</td></tr>
<tr><td></td><td colspan="16">Pure code relocatable value</td><td>R = 5</td></tr>
<tr><td></td><td>F</td><td colspan="11">MPY/Symbol ID number $\boxed{1}$</td><td colspan="4">$\boxed{2}$ MR/MPY</td><td rowspan="2">R = 6[†] (2-wd. add.)</td></tr>
<tr><td></td><td colspan="16">Unrelocated value or offset</td></tr>
<tr><td></td><td colspan="16">SAVE data relocatable value</td><td>R = 7</td></tr>
<tr><td></td><td colspan="4">9th R</td><td colspan="4">10th R</td><td colspan="4">11th R</td><td colspan="4">12th R</td><td></td></tr>
<tr><td></td><td>I</td><td colspan="4">Opcode</td><td colspan="2">&lt;Zero&gt;</td><td colspan="4">MPY</td><td>M</td><td colspan="4">MR</td><td rowspan="2">I: indirect
R = 8 (mem. ref.)</td></tr>
<tr><td></td><td colspan="16">Unrelocated value or offset</td></tr>
</table>

Figure H-11. Extended DBL record (XDBL)

XDBL (continued)



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Symbol ID number											MR			
Unrelocated value or offset															LR
I	Opcode				Symbol ID number										
<Zero>							MPY				M	MR			
Unrelocated value or offset															
I	Symbol ID number											MR			
Actual count								Call sequence				PCAL type			
Unrelocated value or offset															
I	Symbol ID number											MR			
<Zero>												lbl type			
Unrelocated value or offset															
F	MPY/Symbol ID number 1											2 MR/MPY			
Unrelocated value or offset															

R = 9 (Byte addr.)
LR: 0 - left
1 - right

R = 10
M: 1 - immediate
0 - mem. ref.

R = 11
PCAL: 0 - constant
1 - variable

R = 12
Call seq.: 0 - unspec.
1 - .ENTR
2 - .ENTN

R = 13 † (DLD=J)
lbl type: 0 - unspec.
1 - internal
2 - extrenal

MPY:
signed 4-bit integer

†For R6 and R13:

```
If F=1 then
  high 4 bits of field 1 is MPY
  field 2 is MR
  symbol ID=0
else if field 1 = 0 then
  field 2 is MPY
  field 1 is symbol ID => MR
else
  symbol ID = 0
  MPY = 1
  MR = field 2
```

Figure H-11. Extended DBL Record (XDBL) (continued)

RPL

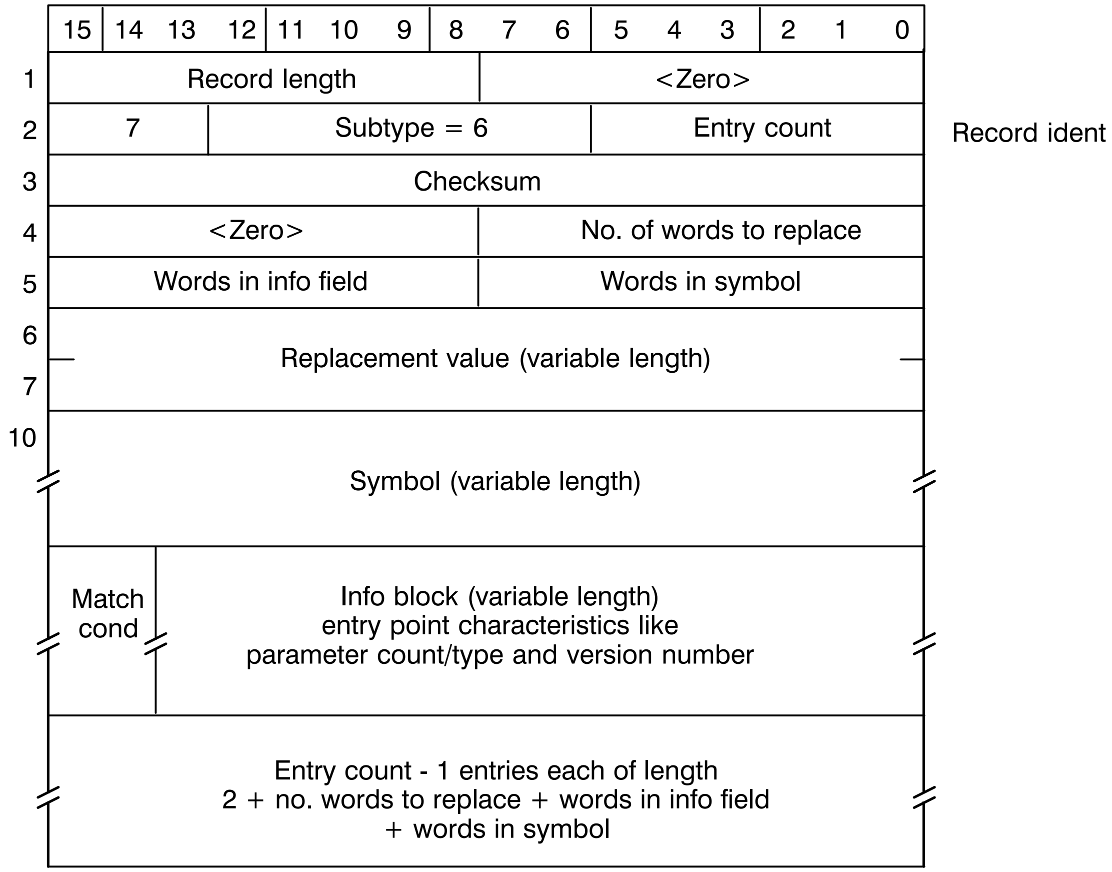

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
1
Record length
<Zero>
2
7
Subtype = 6
Entry count
Record ident
3
Checksum
4
<Zero>
No. of words to replace
5
Words in info field
Words in symbol
6
7
Replacement value (variable length)
10
Symbol (variable length)
Match cond
Info block (variable length)
entry point characteristics like
parameter count/type and version number
Entry count - 1 entries each of length
2 + no. words to replace + words in info field
+ words in symbol

Figure H-12. RPL Record

END

<table>
<tr><td></td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8"><Zero></td></tr>
<tr><td>2</td><td colspan="3">5</td><td colspan="10"><Zero></td><td colspan="2">R</td><td>T</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td></tr>
<tr><td>4</td><td>0</td><td colspan="15">Unrelocated transfer address</td></tr>
</table>

T:
0 - no xfer.add.
1 - xfer. add.

R:
0 = program
1 = base page
2 = common
3 = absolute

Figure H-13. END Record

XEND

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8">&lt;Zero&gt;</td><td></td></tr>
<tr><td>2</td><td colspan="3">7</td><td colspan="7">Subtype = 5</td><td colspan="6"></td><td>Record ident</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td></td></tr>
<tr><td>4</td><td colspan="16">Checksum of checksums</td><td></td></tr>
<tr><td>5</td><td colspan="12">&lt;Zero&gt;</td><td colspan="4">Trans type</td><td></td></tr>
<tr><td>6</td><td>0</td><td colspan="11">Symbol ID number</td><td colspan="4">MR</td><td></td></tr>
<tr><td>7</td><td colspan="16">Unrelocated value or offset</td><td></td></tr>
<tr><td>8</td><td colspan="16"></td><td></td></tr>
</table>

MR:
- 0 - absolute
- 1 - program
- 2 - base page
- 3 - common
- 4 - pure code
- 5 - local EMA
- 6 - SAVE area

Trans type:
- 0 - no add.
- 1 - tran. add.

Figure H-14. Extended END Record (XEND)

GEN

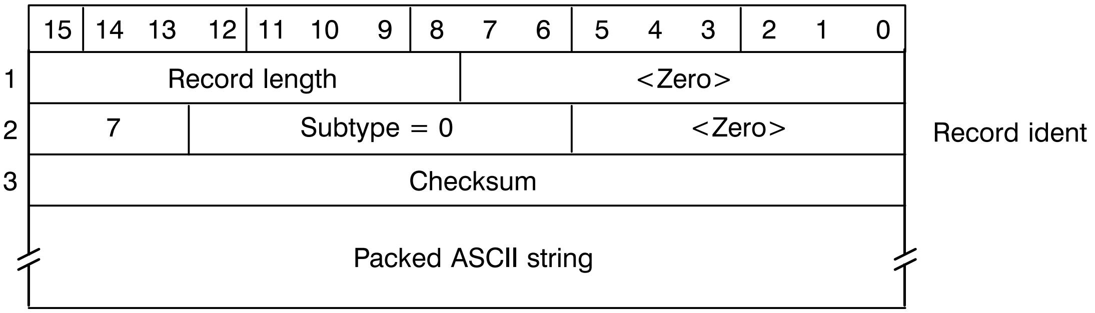


Figure H-15. GEN Record

LOD

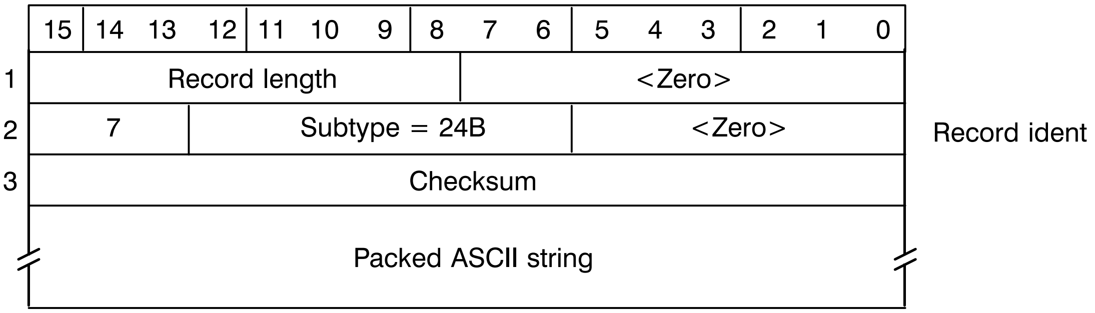


Figure H-16. LOD Record

DEBUG

DEBUG line number record:

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8">&lt;Zero&gt;</td><td></td></tr>
<tr><td>2</td><td colspan="3">7</td><td colspan="7">Subtype = 12B</td><td colspan="6">0</td><td>Record ident</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td></td></tr>
<tr><td>4</td><td colspan="8">&lt;Zero&gt;</td><td colspan="8">Number of chunks</td><td></td></tr>
</table>

Each entry or chunk has the following format:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="16"></td></tr>
<tr><td colspan="16">Starting line number</td></tr>
<tr><td colspan="4">0</td><td colspan="4">Substmt no.</td><td colspan="8">Offset count + 1</td></tr>
<tr><td colspan="12"></td><td colspan="4">MR</td></tr>
<tr><td colspan="16">Relocatable address of first line</td></tr>
<tr><td>s</td><td colspan="7">Offset</td><td>s</td><td colspan="7">Offset</td></tr>
<tr><td>s</td><td colspan="7">Offset</td><td>s</td><td colspan="7">Offset</td></tr>
</table>

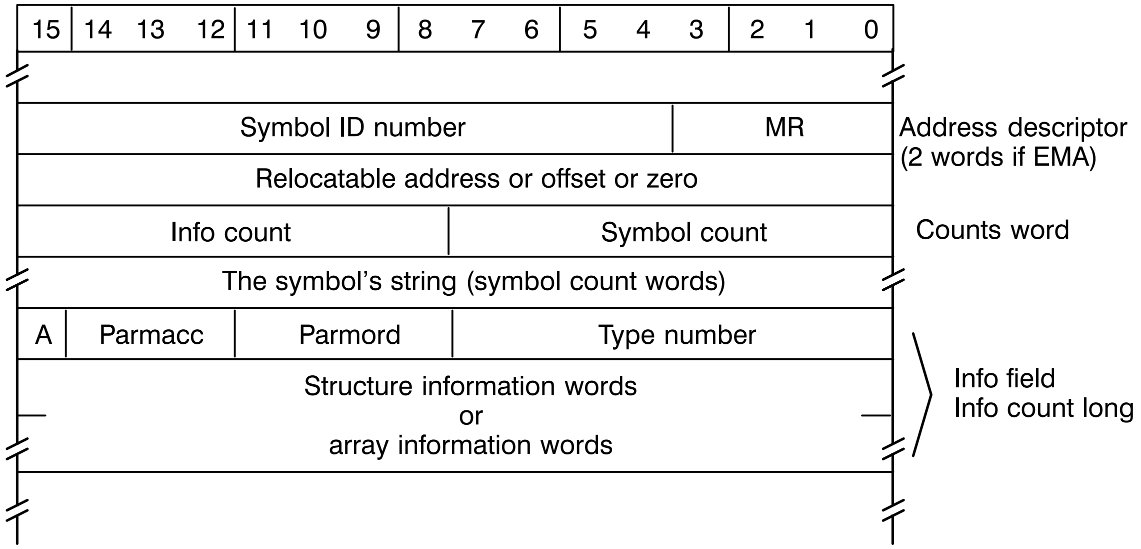
DEBUG symbol table record:

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8">&lt;Zero&gt;</td><td></td></tr>
<tr><td>2</td><td colspan="3">7</td><td colspan="7">Subtype = 12B</td><td colspan="6">1</td><td>Record ident</td></tr>
<tr><td>3</td><td colspan="16">Checksum</td><td></td></tr>
<tr><td>4</td><td colspan="16">Entry count = no. of entries in this record</td><td></td></tr>
</table>

Figure H-17. DEBUG Record

DEBUG (continued)

Each entry has the following format:



DEBUG array sub part:

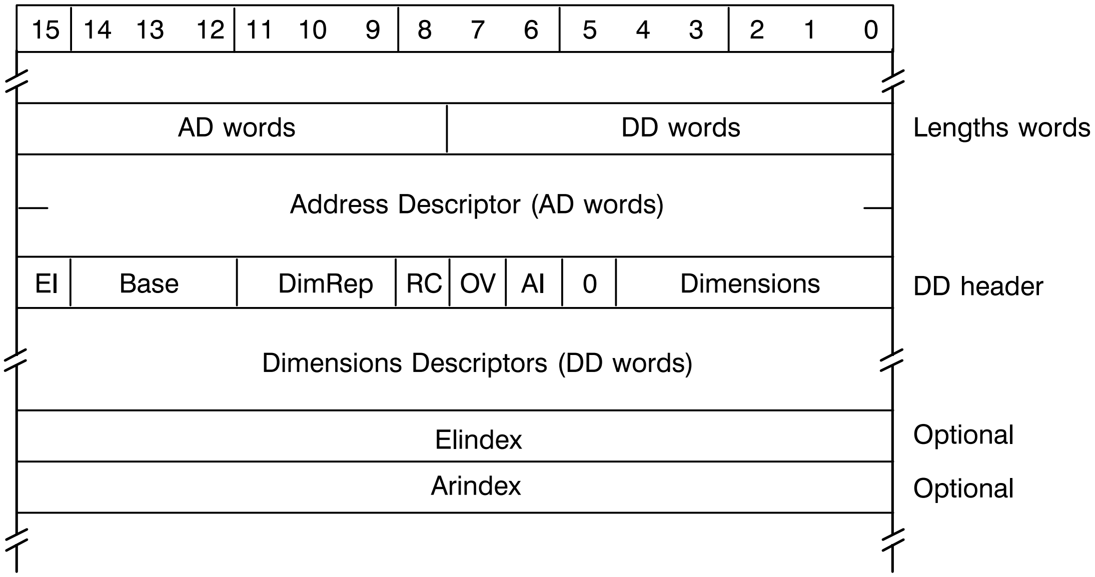


Figure H-17. DEBUG Record (continued)

LINDEX

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td colspan="8">Record length</td><td colspan="8"><Zero></td></tr>
<tr><td>2</td><td colspan="3">7</td><td colspan="7">Subtype = 44B</td><td colspan="6"><Zero></td></tr>
<tr><td>3</td><td colspan="16">Checksum</td></tr>
<tr><td>4</td><td colspan="16">Sub type</td></tr>
</table>

Record ident

Sub type:
0 - fill record
−1 - ptr. to index
−2 - end of lib.
>0 - index record

For the various sub types we have the following:

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td></td><td colspan="16"></td></tr>
<tr><td>4</td><td colspan="16">Sub type −1</td></tr>
<tr><td>5</td><td colspan="16">Block number of first index rec.</td></tr>
<tr><td>6</td><td colspan="16">0</td></tr>
</table>

(First record in the file)

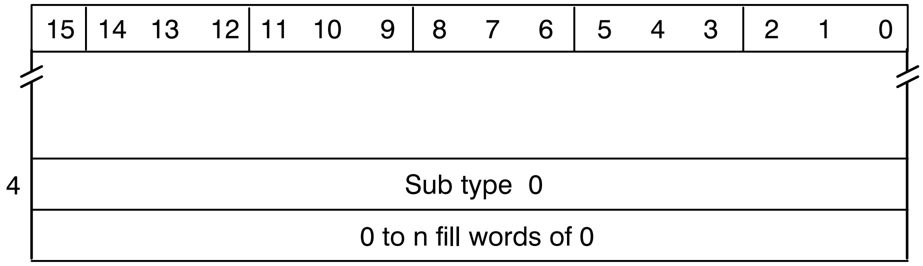
<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td></td><td colspan="16"></td></tr>
<tr><td>4</td><td colspan="16">Sub type −2</td></tr>
<tr><td>5</td><td colspan="16">0</td></tr>
<tr><td>6</td><td colspan="16">0</td></tr>
</table>

(This record seems to have no purpose)

(Appears after last module)

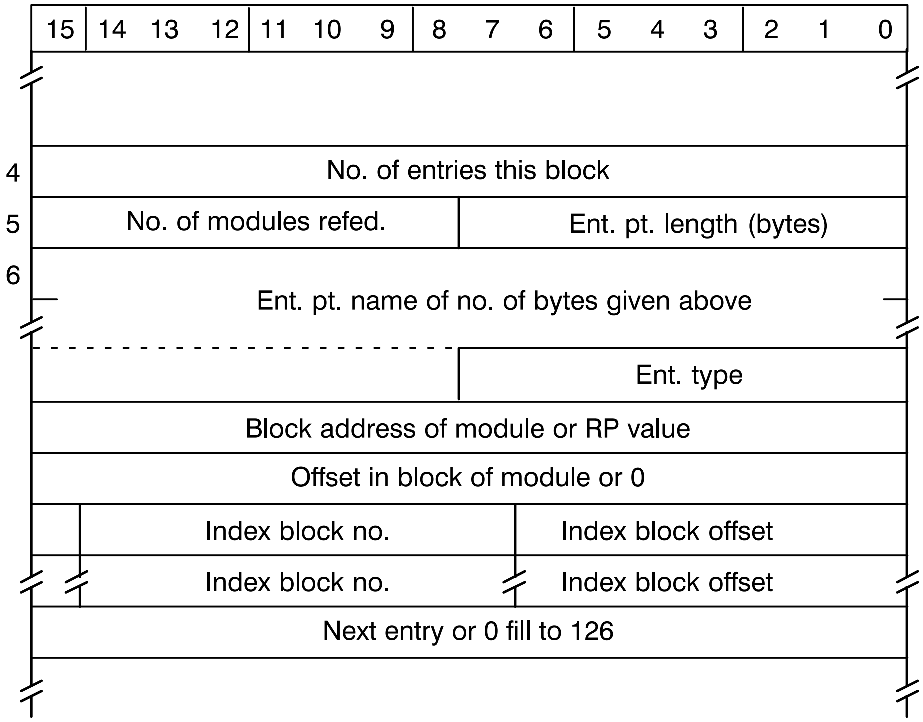
Figure H-18. LINDEX Record

LINDEX (continued)



(This record fills out the last library block so that the index will start on a block boundry)

This index itself:


15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
4
No. of entries this block
5
No. of modules refed.
Ent. pt. length (bytes)
6
Ent. pt. name of no. of bytes given above
Ent. type
Block address of module or RP value
Offset in block of module or 0
Index block no.
Index block offset
Index block no.
Index block offset
Next entry or 0 fill to 126

(sub type >0)
If ent. pt. length ent. type fills odd byte, else it is low part of next word.

Ent. type:
1 - std. entry
4 - RPL
5 - allocate data

No. of modules refed. of these (may be 0)

Figure H-18. LINDEX Record (continued)

INDXR

Index Pointer Record:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	Record length								<Zero>								
2	7			Subtype = 44B							<Zero>						Record ident
3	Checksum																First record in an Indxr indexed file.
4	Record number of index																
5	Block number of index																
6	Offset in block of index																

For the index itself we have the following:

Index Module Record:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	Record length								<Zero>								
2	7			Subtype = 44B							<Zero>						Record ident
3	Checksum																
4	1	No. words in name															The sign bit indicates a module record. Following will be the entry points for this module.
	Module name variable length																
	Record number of module																
	Block number of module																
	Offset in block of module																

Figure H-19. INDXR Record

INDXR (continued)

Index entry point record:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	Record length								<Zero>								
2	7			Subtype = 44B							<Zero>						Record ident
3	Checksum																
4	0	No. words in entry point symbol															The sign bit = 0 indicates an entry point record. Several may be in the same record.
	Entry point name variable length																
	The above length and symbol words are repeated as needed																

Figure H-19. INDXR Record (continued)

Implementation Notes

The following tasks are performed on each pass of MACRO.

Pass 1: (macro pass)

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: (assembly pass 1)

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. MIC instruction is processed, so the length of any user-defined instruction is known.
6. The file produced on Pass 1 is left for Pass 3 unchanged.
7. The symbol table is preserved for Pass 3.

Pass 3: (assembly pass 2)

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: (cross reference pass)

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.

Pass 5:

This pass is optional, depending on whether the user requires absolute assembly or not.

1. Convert the relocatable records produced into an absolute form. This must be done since absolute programs are produced in the same way relocatables are.

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/1000 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/1000 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 Opcodes

The pseudo opcodes 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/1000 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 100      ; Reserve 100 words on base page.
       :
      ORR          ; Continue main program.
       :
      ORB          ; Resume assignment at next available location
       :           ; on 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.

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 opcodes 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 opcodes 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 neither letter is included in the control statement, the related set of statements appears on the assembler output listing if the LIST LONG option is used. However, these statements are 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 codes 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.

A more general way of doing conditional assembly is through the AIF statement. The selection may be done in the runstring by initializing the system runstring 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 &.RS1)

  NAM TEST
   :
  AIF &.RS1 = 'N'
   sequence of statements
  AENDIF
   :
  END
```

REP

Syntax:

[*label*] REP *n* [;*comments*]

The REP pseudo opcode 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. *n* may be any absolute expression. Comment is ignored and the instruction following the comment is repeated.

A label specified in the REP pseudo opcode is assigned to the first repetition of the statement. A label should not be part of the instruction to be repeated, as 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 nth word or 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.
```

Organization of Common Block

PROG1	PROG2	Common
name	name	block
ADDR1	AAA	location 1
		location 2
	AAB	location 3
		location 4
	AAC	location 5
ADDR2	AAD	location 6
		location 7
		location 8
		location 9
		location 10
ADDR3		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 opcode. 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

The new way to declare EMA space is to use the ALLOC EMA statement.

Syntax:

label `EMA` *m1*`,`*m2* `[;`*comments*`]`

The EMA pseudo opcode defines an extended memory area (EMA) where *m1* is the EMA size in pages and *m2* is the mapping segment (MSEG) size in pages. Operands *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 *m2* evaluates to 0, the maximum possible size for MSEG will be assigned at load time.

The EMA pseudo opcode can be used only in a relocatable program. Only one EMA pseudo opcode is allowed.

THE EMA COMMAND CANNOT BE USED IN THE SAME PROGRAM WITH A RELOC EMA COMMAND OR AN ALLOC EMA COMMAND. An EMA pseudo opcode 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 opcodes.

References to EMA labels are processed at load time as indirect addresses through a base page link. 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 twenty-page EMA that has a five-page mapping segment. The program first calls the EMAST subroutine to return the needed EMA information. The program then calls MMAP to map the third MSEG into its logical address space. The address of the starting logical page of MSEG is then converted into a word address. Lastly, the program stores the value at the start of the third MSEG into word 1028 of the third MSEG and terminates. Refer to Figure J-1 for a pictorial explanation of the elements that are being addressed.

```
      NAM EMAPR,3
      ENT EMAPR
      EXT MMAP
      EXT EMAST
EMALB EMA 20,5 ; 20 pages of EMA, 5 pages per mapping segment (MSEG
*
* Call MMAP to map third MSEG into programs logical address space.
*
EMAPR JSB EMAST
      DEF RTN
      DEF NEMA  ; Total size of EMA (returned).
      DEF NPGS  ; Total size of MSEG (returned).
      DEF IMSEG ; Starting logical page of MSEG (returned).
*
* Call EMAST to return information about the program’s EMA.
*
RTN   JSB MMAP
      DEF RTN2
      DEF IPGS  ; Offset in pages of MSEG being mapped.
      DEF NPGS  ; Number of pages in MSEG.
RTN2  LDB IMSEG
      BLF,BLF   ; Convert the page address into a word address.
      BLS,BLS
      LDA @B ; Load A-Reg with the address of the 1st word of MSEG.
      ADB =D1027
      STA @B ; Store A-Reg into the 1028th word of current MSEG.
      JSB EXEC  ; Terminate program.
      DEF *+2
      DEF =D6
IPGS  DEC 10
NPGS  BSS 1
IMSEG BSS 1
NEMA  BSS 1
      END EMAPR
```

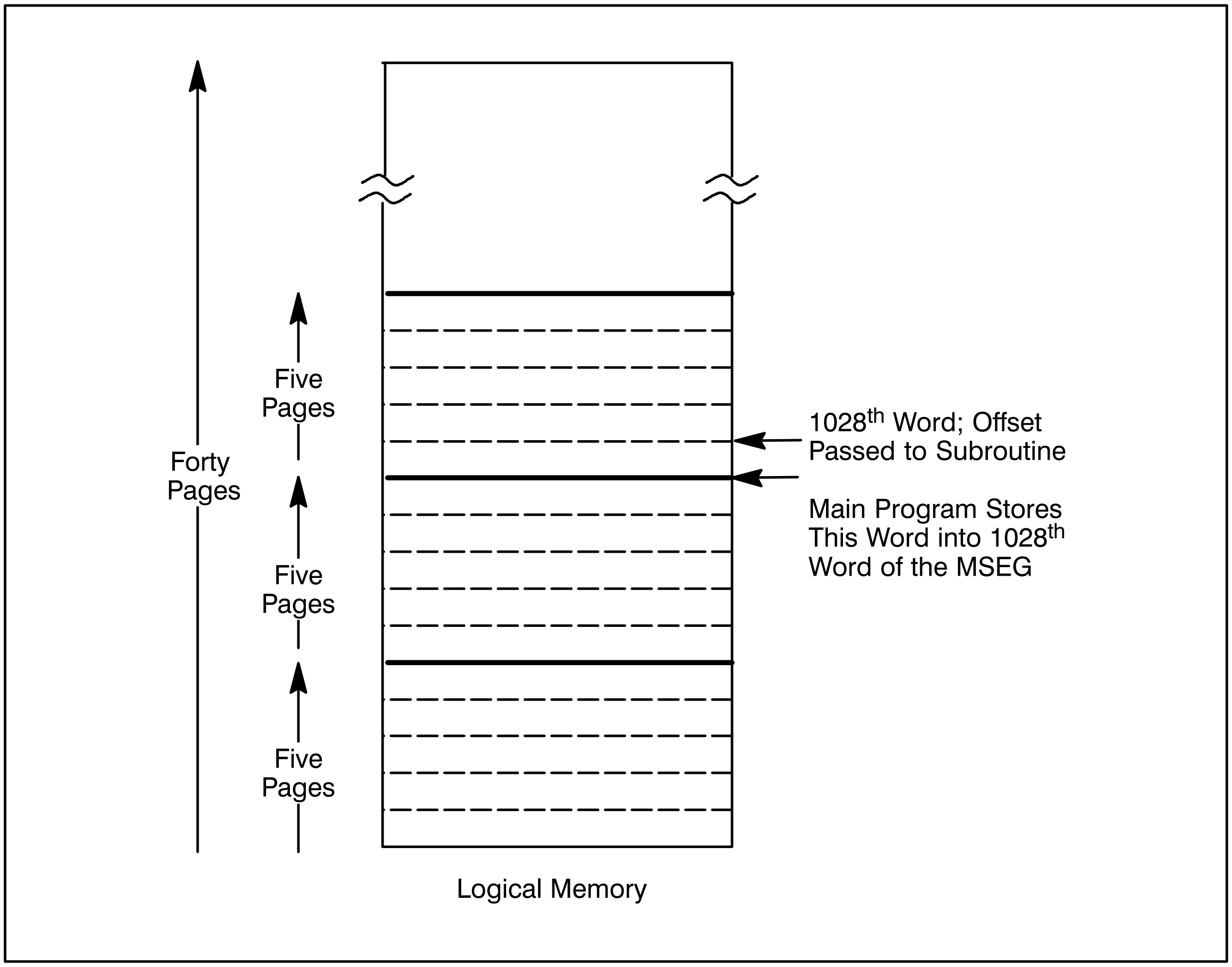

Forty Pages
Five Pages
Five Pages
Five Pages
1028th Word; Offset Passed to Subroutine
Main Program Stores This Word into 1028th Word of the MSEG
Logical Memory

Figure J-1. Pictorial Explanation of Elements Being Addressed

UNL

Syntax:

```
UNL [;comments]
```

List output is suppressed from the assembly listing, beginning with the UNL pseudo opcode and continuing for all instructions and comments until either an LST or an END pseudo opcode 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 [;comments]
```

The LST pseudo opcode 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/1000 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 after 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 opcode 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 opcode 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 opcode 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 opcode 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 101*xxx* or OCT 105*xxx* statement (where *xxx* is 140 through 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 105*xxx* statement.

With the MIC pseudo opcode, 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.

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 opcode. The RAM pseudo opcode generates an executable machine instruction which when executed will cause a jump to microcode. The high order bits of the instruction are 105 octal and low order bits are 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
```

generate this octal object code:

```
     105016
```

K

System Assembly-Time Variables

The assembler declares system assembly-time variables (ATVs) whose values are available to you, and in some cases, they can be altered. 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.

The following are the system assembly-time variables:

```
&.Q              &.RS1              &.REP
&.ERROR          &.RS2              &.PCOUNT
&.PARM[ ]
 &.DATE (6-character date in form YYMMDD)
```

&.Q

&.Q is a unique number of type integer. It is local to the macro within which it is used and contains a unique number for each separate macro. 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, leading zeros are suppressed when substitution is done. For instance, the example above could generate a label ‘L72’ but not ‘L0072’.

&.ERROR

&.ERROR contains the assembly error count and is a 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. 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; this 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 opcode MNOTE will increment &.ERROR automatically and print out an ASCII string in the program listing. See the description of MNOTE in Chapter 4.

&.DATE

&.DATE is a type character ATV that represents the date. The six-character variable takes the form yymmdd.

&.RS1 and &.RS2

&.RS1 and &.RS2 are type-character global assembly-time variables. They can be used as optional parameters in the MACRO runstring as follows:

```
CI>  MACRO,PROG.MAC,–,–,,,&.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 counted.

For 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.

&.PARM [ ]

&.PARM [*n*] refers to the *n*th parameter in the current macro expansion. *n* may be any legal (system conforming) Assembly Time Variable (ATV) array index. It is an error if *n*<0 or *n*>&.PCOUNT. If the label parameter is present, it is one; otherwise, the first parameter is the one following the macro name.

Following is an example, using &.PCOUNT and &.PARM:

```
     MACRO
      Call &WHO,&P1,&P2,&P3,&P4,&P5,&P6,&P7,&P8,&P9,&P10
     AIF &.PCOUNT = 0
      MNOTE Call WHO??
     AENDIF
      EXT &WHO
      JSB &WHO
      DEF *+&.PCOUNT
&I    ILOCAL 2  ; First Call Index
      AWHILE &I <= &.PCOUNT
       DEF &.PARM[&I]
&I     ISET &I+1
      AEND WHILE
       END MAC
```

HP 1000 Macro Library

This appendix describes the macros available in the HP 1000 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 HP 1000 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 HP 1000s: 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.

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 touch X or Y.

The following is a list of all system macros described in this appendix in order of appearance:

ENTRY
EXIT
CALL

IF
ELSE
ELSEIF
ENDIF

ADD
SUBTRACT
MAX
MIN

SETBIT
CLEARBIT
TESTBIT
FIELD

ROTATE
ASHIFT
LSHIFT
RESOLVE

TEXT
MESSAGE

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,1
 2            nam mvw,7,99 FORTRAN-callable .MVW &.date
 3           maclib $MACLB
 4           GLOBALS
 5  *
 6  * Routine to allow FORTRAN (or pascal, etc.) to use the
 7  * microcoded .MVW word-mover.  Parameter 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 $MACLB.

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.

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 carat (^) 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 opcode. 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 (MACRO) 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/1000:

`LIST short` lists only the macro calls.

`LIST medium` includes the generated code and the calls to macros internal to the library.

`LIST long` gives the complete text of all macros.

Descriptions of System Macros

In the following descriptions, macro names are in uppercase, parameters are in italics, optional parameters are in brackets. Three dots indicate 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
```

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.

For example, following the previous example for ENTRY, the statement:

```
    EXIT
```

generates:

```
    JMP @InitializeData
```

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.)

operand1 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.

operand2 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.

Comparison	Alpha	Sym	Register changed?
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

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 the 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 of the instruction to which an IF, ELSE or ELSEIF jumps. The ENDIF macro does not generate any code, but causes a return to the next outer nesting level.

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
```

In the above example, the end-of-file test is made only if the error code is not negative. Nesting is limited to ten levels; ELSEIF can be used 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 is 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 is 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, and 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 *bitnumber* 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, and 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
```

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 is 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 from 0 to 15. Bit 0 is the least significant, and 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 is used for the operation in any case.

startbit is a number from 0 to 15; bit 0 is the least significant bit, and 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 only 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, and 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 is 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, and 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 *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 opcode 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 (WRITF, 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 a 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
```

M

CDS Assembly Language Programming

Introduction

This appendix contains some suggestions for the assembly language programmer who wishes to use the HP 1000 features commonly called CDS (code and data separation) programming. These features include, as well as the separation of code and data, an improved local variable scheme that supports recursion and reentrance, plus a code segmentation scheme that allows greater speed and flexibility than previous methods. It is necessary, however, that a minimal enhancement to existing assembly code be performed to make optimal use of these features.

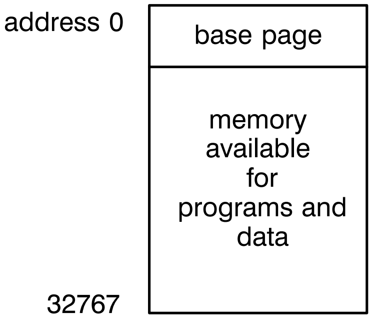
The first part of this appendix contains a brief description of what the CDS architectural enhancements mean to the assembly language programmer, and introduces some terms to be used later. For more advanced issues, it is suggested that you consult the hardware manuals and the programmer's reference manual for your particular HP 1000 computer model number.

The next part of this appendix contains descriptions of the assembly language constructs relevant to the CDS programmer.

The rest contains some examples, and outlines some useful macros supplied by Hewlett-Packard to simplify the programmer's job.

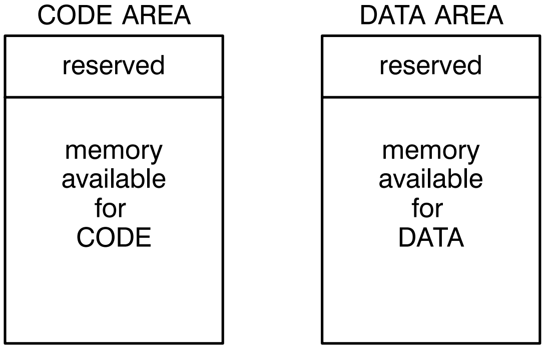
A Brief Outline of CDS Features

To the assembly language programmer without CDS features, the HP 1000 target machine might appear something like this:



In this model, the programmer sees an area of 32767 words available for program space. One page, or 1024 words, of that is required for a base page, and some of the rest may be required by the operating system. The main feature of this model is that the program's code and data are intertwined together into this memory area. This means, for example, that the local variables for a subroutine reside often within a few addresses of the instructions that manipulate them. It also means that code can accidentally get altered even though the programmer thinks data is being manipulated. In addition, in a segmented program the data for the subroutines in a segment can get destroyed when a new segment is brought into memory.

In the CDS model of memory, the programmer sees two separate areas of 32767 words – one for code and one for data:



The programmer can think of the CODE area as one that contains values that don't change (like instructions and some constants), and the DATA area as one that contains values that may change. The HP 1000 CPU knows that instructions that affect data (like LDA or STA) should use the DATA area when they are executed, but instructions that affect the CODE area (like JMP or SSA) should be performed on addresses in the code space. Note that any CODE that is self-modifying must be executed in the DATA space. This includes old-style HP 1000 subroutines that are accessed by the JSB instruction.

It is the programmer's responsibility to decide which parts of the program should be put into data and which into code. This document may help a little but is meant to supplement what you will find in the appropriate programmer's reference manual. This appendix outlines some of the assembly language commands that tell the assembler where the programmer wants these parts of the program placed.

Assembly Language Constructs

This section describes some of the opcodes and pseudo opcodes available to the CDS programmer. Some examples appear in the next section.

Syntax of Commands

`CDS` *keyword*

where:

keyword is ON or OFF

`PCAL` *SubName*`,` *ParameterCount*`,` *CallingSequence* *PcalType*

where:

SubName is the name of the subroutine.

ParameterCount is the number of parameters passed.

CallingSequence is
0: unspecified
1: .ENTR
2: .ENTN
3: standard PCAL

PcalType is
0: Normal
1: procedure passed as param

`RELOC` *keyword*

where:

keyword is CODE, DATA, LOCAL, STATIC, PROG, COMMON, SAVE, EMA, or BASE.

`LABEL` *ProcName*

where:

ProcName is the name of the procedure referenced.

`BREAK ...` (no operand)

CDS Command

The CDS command is a pseudo opcode that instructs the assembler that your code is intended to run on a CDS machine. This command should appear in every module of your program that will be loaded into the CDS environment. The command takes one operand:

```
    CDS ON ; comments
```

If a label appears on this command, it is ignored. The keyword operand ON (lowercase or uppercase is allowed) enables all of the CDS features of the assembler.

The CDS command must appear in your source before any opcodes or pseudo opcodes that generate code or data. It is suggested that you place this command immediately after the NAM statement in your source program.

PCAL

The PCAL opcode calls a subroutine. When in CDS mode, the JSB, .ENTR, .ENTP, .ENTN, and .ENTC instructions should never be used. The PCAL instruction, when executed, sets up a CDS standard call to the subroutine specified, handles all of the parameters passed, and allocates memory for local variables (in what is called an “activation record”). The command takes four parameters. Only the first one is required.

```
    PCAL SubName, #Parameters, CallingSequence, PcalType
```

where:

SubName is the name of the subroutine being called.

#Parameters is the count of the number of parameters passed.

CallingSequence is an integer value describing the type of PCAL to make:

0 – unspecified Standard calling sequence filled in by LINK.

1 .ENTR – The .ENTR form of PCAL emulates old .ENTR calls for routines in the data segment. This is for calling old subroutines that you have not converted to take advantage of CDS features.

2 .ENTN – The .ENTN form of PCAL emulates old .ENTN calls for routines in the data segment.

3 PCAL – This is the standard calling sequence when you are calling from a CDS subroutine to another CDS subroutine. It is expected that most PCAL instructions you use will contain this flag.

PcalType When the address of the subroutine to be called is passed to the calling subroutine as a parameter (that is, a “variable procedure”), this flag is 1. Otherwise, it is 0.

It is strongly suggested that you use PCALL, the macro supplied by HP, to perform subroutine calls in your code. This command has a simpler set of rules and allows a greater flexibility. (Refer to “PCALL” later in this appendix.)

Parameter passing is done by following the PCAL instruction with a list of DEFs to the parameters to be passed. This imposes a stricter set of calling sequence rules than offered by the JSB instruction. All parameters must be passed in this manner. The called subroutine cannot get the value of anything passed it by simply loading through the return address indirect.

Example:

```
    FORTRAN:                 Non-CDS style:       CDS style:

Call Subr(P1, P2, P3)           Jsb Subr         PCAL Subr,3,3,0
                                def *+4          def P1
                                def P1           def P2
                                def P2           def P3
                                def P3
```

Again, use the PCALL macro to make parameter passing a little easier than this.

Example Macro of ENTRY with PCAL

```
macro,m,l,t
*       ENTRY macro comments
*       Brief Description:  This macro sets up a subroutine that
*         will be called using the standard calling format for
*         CDS (PCALL SUBR address, DEF P1, DEF P2, etc.).
*         It gives the subroutine formal parameters names.  The
*         CDS hardware automatically fills the actual parameter
*         value in at run time.
*       Registers Affected:  all are clobbered
*       Unusual Side Effects/ Miscellaneous Notes: None so far
*       Parameters:
*         &NAME is the subroutine name
*         &FP1,&FP2,...,&FP10 are the formal parameters of the
*         subroutine
*       Alternate Calling Formats/ Default Parameters:  May be
*         called with 2 to 10 of the macro parameters &FP1 to
*         &FP10; the ones not used will default to ’’.
*
        MACRO
&NAME   ENTRY &FP1,&FP2,&FP3,&FP4,&FP5,&FP6,&FP7,&FP8,&FP9,&FP10
        AIF :T:&.LocalMacroUsed <> ’U’
           AIF &.LocalMacroUsed <> ’Used last on END’
             MNOTE ’The LOCAL statement should not be used before
                     ENTRY’
             AENDIF
        AENDIF
        RELOC LOCAL
        ENT &NAME
&NAME       DEF !#!LocalCount
            BSS 5
        AIF &FP1 <> ’’
&FP1        NOP
        AENDIF
        AIF &FP2 <> ’’
&FP2        NOP
        AENDIF
        AIF &FP3 <> ’’
&FP3        NOP
        AENDIF
        AIF &FP4 <> ’’
&FP4        NOP
        AENDIF
        AIF &FP5 <> ’’
&FP5        NOP
        AENDIF
        AIF &FP6 <> ’’
&FP6        NOP
        AENDIF
        AIF &FP7 <> ’’
&FP7        NOP
        AENDIF
        AIF &FP8 <> ’’
&FP8        NOP
        AENDIF
        AIF &FP9 <> ’’
```

```
&FP9        NOP
        AENDIF
        AIF &FP10 <> ’’
&FP10       NOP
        AENDIF
        ENDMAC
*
*       EXIT macro comments
*       Brief description:  This macro exits a subroutine that has
*         been entered with the ENTRY macro.
*       Registers Affected: none
*
        MACRO
        EXIT
            OCT 105417      ; octal instruction for EXIT

*       LOCAL macro comments
*       Brief description: This macro is used to declare the names
*         and sizes of the local variables in a CDS user’s program.
*         It is used both to assure that the users will not try to
*         initialize their variables  (either overtly like
*         ’FOO DEC 12’ or accidently like ’FOO NOP’) and to assure
*         that no locals precede the ENTRY macro.  This is done via
*         an assembly time variable.
*       Only data is generated.
*       The current relocation space counter is modified to be
*         LOCAL.
*
        MACRO
&Lname  LOCAL &size
        AIF :T:&.LocalMacroUsed = ’U’
&.LocalMacroUsed CGLOBAL ’Used last on LOCAL’
        AELSE
&.LocalMacroUsed CSET    ’Used last on LOCAL’
        AENDIF
        RELOC LOCAL
&Lname  bss &size
        ENDMAC

*       END macro comments
*       Brief description:  Used only to set up the variable
*         !#!LocalCount to contain a count of the number of local
*         words used by the program.  This is put into the first
*         word of the code via the ENTRY macro.

        MACRO
         END
         RELOC LOCAL
!#!LocalCount equ *
         AIF :T:&.LocalMacroUsed <> ’U’
&.LocalMacroUsed CSET ’Used last on END’
         AENDIF
         :OP:END
        ENDMAC
        END
```

RELOC Command

The RELOC command instructs the assembler as to whether the opcodes following it are to be assembled into the code space, data space, or are local or static variables. It is with this command that the separation of code and data can be achieved. Although there are a total of nine keywords legal on this command, only four are relevent to CDS programming:

CODE instructs the assembler that all of the instructions following are to be assembled into the code space.

DATA instructs the assembler that the following values are to be put into the data space.

LOCAL defines local variables for your subroutine. The number of words must not exceed 1024 words. All values placed in variables that reside in the LOCAL area are lost when the subroutine is exited, that is, they are not preserved from one call to the next.

STATIC is also used to define local variables. However, the value of variables put into this space is preserved from one subroutine call to the next.

When CDS is turned on, the keywords PROG, COMMON, SAVE, and EMA are still legal. PROG is the same as DATA in CDS programming, and BASE is illegal.

For examples of RELOC, refer to "Program Examples" later in this appendix.

LABEL Command

The LABEL pseudo opcode allows "variable procedures" calls. This opcode is present for completeness but is not expected to be used by the typical assembly language program. A variable procedure is one whose label has been passed to the current procedure as a parameter. Refer to "Example of LABEL Statement" later in this appendix.

```
LABEL VarParam
```

where:

VarParam is the name of a procedure.

The LABEL pseudo opcode generates a one-word label that may be used with a PCAL opcode where PCAL is set to 1.

BREAK Command

The BREAK pseudo opcode indicates those parts of your program at which natural breaks in the flow of the program occur. It is used by the loader to construct current page links for off-page references. This instruction must be used at least every 511 words of code. Here are some guidelines:

1. A break should not be placed where it might be executed.
2. An indexed jump must never cross a break.
3. A skip instruction must never cross a break.

There must be a BREAK command every 511 words, it is suggested that you put one approximately every 100 lines of code.

Some Useful Macros

This section contains descriptions of how to use some macros that are provided by HP to make your programming job easier.

What is a Macro?

A macro can be thought of as a new pseudo opcode or opcode that is not normally built into the assembler, one that does something special. By writing macros, the programmer can make a set of customized opcodes. Many of the so-called "opcodes" that are useful to the CDS programmer are really customized macros written for you by HP. You can gain access to these macros by placing the MACLIB command into your code before you use any of the special opcodes. See your system manager for the name of the macro library file that goes with this command.

This macro library can be used with any other macro libraries you may wish to use, but the particular MACLIB command for these special opcodes must appear first.

PCALL

The PCALL macro generates the code to perform a subroutine call. It is the CDS equivalent of the JSB instruction. The PCALL macro allows you to specify the subroutine you wish to call, and the names of the parameters to be passed, all in one line of code. Example:

```
PCALL SubName, P1, P2, P3
```

where:

SubName is the name of the subroutine called.

P1,P2,P3 are the parameters to be passed. Up to 10 parameters can appear on the PCALL macro.

ENTRY

The ENTRY macro marks the start of a subroutine. It is used in place of the JSB .ENTR in non-CDS programs. This macro contains the name of the subroutine and the names of the parameters the subroutine takes. Example:

```
SubName ENTRY X1, X2, X3
```

where:

SubName is the name of the subroutine that is being defined here. Note that the name starts in column 1 of the assembly language source.

X1,X2,X3 are local names that are assigned to the parameters.

No RELOC LOCAL commands or LOCAL macro calls (see below) can be used before this macro.

LOCAL

The LOCAL macro declares the names and sizes of all of the local variables to be used by this subroutine. Remember that the values of these variables are undefined when the subroutine exits, and therefore are not preserved from one call of the subroutine to another. The macro contains the name of a variable and the number of words of memory it should occupy.

Example:

MyVar `LOCAL 2`

In this example, the variable *MyVar* occupies two words in the local variable space. This macro should not appear before the ENTRY macro.

EXIT

The EXIT macro marks a subroutine return. When executed, control is transferred back to the subroutine that called this one. This macro is analogous to JMP *SubName*,I used in non-CDS assembly language programs.

Example:

```
EXIT
```

Program Examples

This section contains examples of assembly language code written to use the CDS features. Wherever possible, the new macros shown above were used.

The following piece of code contains a main program that initializes some global variables and calls some subroutines. The main feature illustrated here is the way subroutines get called and parameters, local and global variables get set up.

General Example

```
      NAM MainProg,4 CDS demo program

      MACLIB '.CDMAC::crn'  ; allows access to CDS macros

      CDS ON                ; reminds the assembler that this is a
                            ; CDS program.
      EXT Subr1, Subr2

; Data declaration section.  Declare some global data to be used
; by other subroutines.

      RELOC DATA   ; assemble the following opcodes in the DATA area

         ENT Glob1
Glob1    BSS 1     ; value to be filled in when program runs
         ENT Glob2
Glob2    DEC 23    ; value initialized at assembly time

Mdata1   BSS 100   ; this data is local to MainProg but will
Mdata2   BSS 200   ; be passed as a parameter

; Data all set up.  Now assemble the following opcodes
; in the CODE area

      RELOC code

MainProg  :        ; program starts here.
          :
         LDA =d10  ; put some values into MainProg's data
         STA Mdata1
         LDA =d20
         STA Mdata2
          :
          :
         PCALL Subr1, Mdata1, Mdata2
          :
         PCALL Subr2, =D23
          :
          :
         END MainProg
         NAM Subr1
         ENT Subr1
         EXT Glob1      ; allows access to the globals

         CDS on

; Start of subroutine.  Declare the names of the parameters first
; and then the names and sizes of the local variables.

Subr1    ENTRY P1, P2   ; local names for Mdata1 and Mdata2 above

SubrL1   LOCAL 2        ; 2 words long; initial value is undefined
SubrL2   LOCAL 1
; Variables are all declared.  Start the code.

           LDA @P1        ; A-Register gets the value 10.
            :
           STA SubrL2     ; access locals normally
            :
           LDA Glob1      ; access globals normally
            :
           EXIT           ; return to caller

         END

         NAM Subr2
         ENT Subr2

         CDS ON

; Declare the locals and the names of the parameters

Subr2    ENTRY Const

Subr2L1  LOCAL 2

; This subroutine will have some local data whose value remains
; preserved from one call to the next.  This goes in the static
; data area.

         RELOC STATIC

Subr2L2   BSS 12

; Start the code

         RELOC code
           :
          LDA @Const        ; A-Register gets the 23 passed above.
           :
          EXIT

          END
```

Example of LABEL Statement

This statement is not commonly used in typical assembly language programs. This opcode is present for completeness, and for any high level language processors that may generate assembly code.

In the following example, the subroutine SQRT is passed as a parameter to the subroutine CALLER, which then calls SIN as a variable procedure.

```
    NAM FatherRoutine
    MACLIB '$CDSLB::crn'
     :
     :
    EXT SQRT
Var LABEL SQRT
     :
    PCALL Caller, Var
     :
    END

    NAM Caller
     :
Caller ENTRY VarParam
     :
    PCAL @VarParam,1,3,1
     :
    END
```

N

CDSONOFF Macro Library

Introduction

This macro library is designed to facilitate writing “good” assembly code that may be used in either the CDS or non-CDS environment. It is written so that all modules (that is, NAM/END pairs) in a file will be compiled with CDS on or off. In other words, the library does not support a file containing some CDS modules along with some non-CDS modules.

The problems addressed by this macro library are:

- Local and global subroutine call and entry sequences including optional parameters
- Alternate returns (p+1, 2, etc.)
- String descriptor definition
- Relocation space changes

In addition, a trace back macro is provided to generate the code needed to identify the module to the FTN run time error trace back facility.

CDS/Non-CDS Differences

Calls

Subroutine call sequences are different in CDS because the JSB (which modifies its target address) cannot be used except to call routines in data space. Therefore, the calls in CDS and non-CDS are different. Likewise, the entry sequences required for the two modes differ. This is further confused if you want to have one or more optional parameters. The code to set up optional parameters is quite different in the two modes.

Local Subroutines (routines within the same NAM/END pair) must be called with JLY, JLA, or JLB in CDS instead of the JSB used in non-CDS routines. This means that their exit code must also be different. Likewise, bumping the return address is different. A further complication arises when parameters are to be passed to local routines. The best way to do this is to use registers, but in some cases, DEFs may be needed. The standard memory reference instruction may not address the DEF because it is in code space. It is accessible, however, with cross map instructions.

The MVW (move words) and the MBT (move bytes) instructions are not available in CDS code space, but there are move word and move byte instructions that are available. On the other hand, the CMW (compare words) and CBT (compare byte) instructions are neither available in CDS code space nor are there available replacements. To handle these problems, move words, move bytes, compare words, and compare bytes macros are provided. The compare instructions work by calling a data space routine to execute the actual instruction.

Data and Strings

In CDS mode, good code uses as few words of STATIC area as possible. This dictates that most working variables should be put in local space and initialized from code space. Since this is a requirement that is not present in non-CDS mode, macros are provided to initialize and set up strings and string descriptors.

Philosophy

This macro library allows you to write code that could be compiled to run in CDS mode or, by using a simple MACRO runtime parameter, compiled to run in non-CDS mode. The requirements of the two modes are different and some errors that are possible in one mode may not matter in the other.

The macros do most error checking for both modes regardless of the mode. This means that once your code is working in one mode, it should be very close to working in the other mode. Most of the macros check everything that is passed and indicate any errors. They also talk to each other so that if, for example, you code the ENTRY for subroutine PUT to expect a JLY call and then code the CALL to use JLB, an error will be produced regardless of which occurs first, the ENTRY or the CALL.

This macro library includes all of the standard macro library ($MACLB), with the exception of those macros that are expanded in scope here. Code written to use those macros need not be changed to use this library, except that the GLOBALS macro should not be called, nor should that library be included with this one.

Macro Call Sequences

Initialization

This library should be included with MACLIB $CDSONOFF. The macro CDSONOFF should be invoked before the first NAM in the source.

```
CDSONOFF { CDSON  } [,maxlocals] [,col1] [,col2] [,col3]
         { CDSOFF }
```

This macro sets up the library to do either a CDS run or a non-CDS run. It also defines some GLOBAL ATVs, calls the standard MACLB GLOBALS macro, and sets up column formatting.

The first parameter is required and must be spelled out in full: CDSON or CDSOFF. This parameter should be either &.RS1 or &.RS2 to allow CDS selection from the MACRO runstring.

maxlocals defaults to 50 and defines the size, the GLOBAL ATV arrays &.LOCALS, &.LOCALD, and &.LOCALE. You should increase this number only after one of these arrays overflows.

col1, *col2*, and *col3* define the three column parameters for the COL and MACRO opcodes. They default to 10, 15, and 31. Columns can be turned off in the program (but not for macro-generated code) by specifying *col1* as 0. Normally macros are expanded using the same columns as the code except that the opcode is indented by 1.

The following ATVs are defined here:

Name	Type	Modifiable	Comments
&.ISCDS	I	NO	1 if CDSON, else 0.
&.SPACE	C	NO	'CM' if current relocation space is COMMON else the first two characters of the current relocation space.
&.COL1	I	YES	current Column specifications
&.COL2	I	YES	for macro calls.
&.COL3	I	YES	

NEWSUB

NEWSUB [*max_parms*]

This macro must be called after each NAM statement and before the first code in the module. It resets GLOBAL ATVs used to track subroutine calls and LOCAL space usage.

max_parms, if coded, defines the highest number of parameters to expect on subsequent ENTRY and OENTRY calls in this module. If it is not coded, the first occurrence of ENTRY, OENTRY, or END defines the amount of LOCAL space required (in CDS mode) for parameters. Local declarations are deferred until the first ENTRY, OENTRY, or END. This means that local labels are not defined until after the deferral and could cause errors in the assembly pass if any of these local variables are referred to before definition in a context that requires them to be defined.

The cases where *max_parms* should be coded are as follows:

- There are no OENTRY or ENTRY calls. Presumably, this is a main and *max_parms* can be set to 0.
- An ENTRY or OENTRY other than the first requires more parameters than the first ENTRY or OENTRY.
- Code prior to the first entry requires LOCAL labels to be defined.

If *max_parms* is coded and is too small, an error is noted by ENTRY or OENTRY. If exactly *max_parms* parameters are not required by at least one OENTRY or ENTRY call in the module, an error will be noted by the END macro.

TRACEBACK

`TRACEBACK` *name*

If used, this macro should be called immediately after the NEWSUB macro. *name* should be the module name or primary entry point name. This macro does nothing in non-CDS mode. In CDS mode, it generates code that allows the FORTRAN run time error routine to identify and print the module name if it is in the call sequence that resulted in the run time error. If the module and those it calls do not reference the FORTRAN run time error routine (!NFEX or !EXIT), the TRACEBACK macro should not be called. Otherwise, it is recommended. The cost in code space is the name string storage + 7 words. One word is also used at the end of the LOCAL area.

Entry Macros

There are four ENTRY macros:

ENTRY – Entry for calls from other modules. Call with CALL macro.

OENTRY – Same as entry but sets up optional parameters. Call with CALL macro.

DLENTRY – Direct local ENTRY for internal calls from within this module. Call with DLCALL macro.

DENTRY – Direct ENTRY for calls where no return address is passed. Call with DLCALL macro.

ENTRY

name `ENTRY [`*p1*`[, ...[`*p20*`] ...] ]`

name is declared as an entry point for the module that supports the standard FORTRAN calling sequence. Up to 20 parameters can be handled.

If used in CDS mode, LOCAL space is set up and any deferred local definitions are flushed. If TRACEBACK was called, TRACEBACK code is generated to put a name pointer on the stack. If not in CDS mode, a .ENTR call is made to set up the parameter addresses. In either case, the parameter addresses are available under the given names, that is:

LDA *p1* gets the address of the passed-in parameter *p1*.

LDA @*p1* gets its value, etc.

OENTRY

name `OENTRY` *no-optional* `[,`*p1* `...  [,`*p20*`] ...  ]`

This entry is the same as ENTRY above except that extra code is generated to ensure that the last *no-optional* parameter addresses will be zero if they are not supplied. This code is always dependent only on the OENTRY code and thus is reliable in a memory-resident environment, even if the program is aborted within the module.

You can test if a parameter is supplied by looking at the address. If the address is 0, the parameter was not supplied.

For one- or two-word parameters, the following parameter fetch is recommended.

One-word parameter

```
LDA  Default parameter
LDA  @pn   Get parm.  If not supplied, gets A.
```

Two-word parameters

```
DLD  Default
DLD  @pn
```

These sequences depend on the addressability of the A- and B-Registers as memory locations 0 and 1.

DLENTRY

```
name DLENTRY [reg [,exit_reg [,p1....[,p20] ...  ]]]
```

DLENTRY defines a Direct Local ENTRY for *name*. In non-CDS mode, this results in:

```
name NOP
```

In CDS mode, it results in:

```
name EQU *
```

and optionally stores the call register for the EXIT call.

reg is the call register and, if supplied, must be A, B, or Y. *reg* must be supplied if *name* has not yet been called using the DLCALL macro. If *reg* is supplied and there was a prior DLCALL reference to *name*, *reg* must be the same as appeared in the DLCALL.

exit_reg is the exit register. It may be "" (zero-length string), A, B, or any valid expression that defines a memory location to hold the return address.

If *exit_reg* is absent, *reg* is assumed to be the exit register. If *exit_reg* is "–" (a minus sign), a LOCAL is allocated and the call register is saved at that location. For all other cases, the call register is saved at *exit_reg*.

LOADPARMADD

```
LOADPARMADD reg
```

This macro is legal only in a subroutine entered with the DLENTRY macro. It returns in *reg* (A or B) the contents of memory pointed to by the current *exit_reg* (see DLENTRY). If in CDS mode, a cross load is done.

Code generated:

1 word, if not CDS
2 words, if CDS and <exit reg> <> "Y"
3 words, if CDS and <exit reg> = "Y"

DENTRY

```
name DENTRY [p1[, ...[,p20] ...  ]]
```

This macro generates entry sequences for direct calls (see DCALL). In CDS code space, this is the same as ENTRY. If not in code space (that is, DATA space or not CDS), the sequence generated is:

```
P1   NOP
   :
Pn   NOP
name NOP
     ENT name
     EXT .ENTN
     JSB .ENTN
     DEF P1
```

If no parameters are passed, the generated code is:

```
     ENT name
name NOP
```

BUMPEXIT

```
BUMPEXIT
```

The BUMPEXIT macro advances the return address of the subroutine it occurs within. This must be a subroutine entered with ENTRY, OENTRY, DLENTRY, or DENTRY. For LOCAL CDS subroutines (DLENTRY), the exit register is bumped. The macro always generates exactly one word of code.

EXIT, EXIT1, EXIT2

These macros exit (return from) the last subroutine entered with ENTRY, OENTRY, or DLENTRY.

EXIT	exits to the caller at P+1
EXIT1	exits to the caller at P+2
EXIT2	exits to the caller at P+3

The actual exit point is affected by BUMPEXIT also, so that a BUMPEXIT call followed by EXIT2 will exit at P+4.

The EXIT macro generates one word of code except for the case of a Direct Local subroutine call with the Y-Register and returning with it (that is, *exit_reg* is Y).

EXIT1 and EXIT2 generate a variable number of words of code depending on what is required.

```
LOADPARMADD reg
```

This macro is legal only in a subroutine entered with the DLENTRY macro. It returns in *reg* (A or B) the contents of memory pointed to by the current *exit_reg* (see DLENTRY). If in CDS mode, a cross load is done.

Code generated:

1 word, if not CDS
2 words, if CDS and <exit reg> <> "Y"
3 words, if CDS and <exit reg> = "Y"

CALL Summary

CALL – Standard FORTRAN-like call to an external.

LCALL – Standard FORTRAN-like call to an internal routine.

DCALL – Same as CALL but no return address defined.

PCALL – Generate PCALL only. Not recommended since there is no non-CDS form, use CALL.

DLCALL – Direct Local CALL to an internal subroutine.

UCALL – Standard call (with return address defined) to a routine that is known to be RPed.*

DUCALL – Direct call (no return address defined) to a routine that is known to be RPed.*

* "Known to be RPed" applies to calls from CDS code space; thus, UCALL EXEC is legal even though the non-CDS code might run on RTE-6/VM where EXEC is not RPed.

CALL, LCALL

```
CALL name [,p1 ...  [,p20 ] ]
LCALL name [,p1 ...  [,p20 ] ]
```

This call generates the standard FORTRAN call sequence to *name*, passing up to 20 parameters. *name* may be in either CODE or DATA space in CDS mode. For CALL, *name* must not be local (the macro generates an EXT to *name*). For LCALL, *name* must be defined locally.

DCALL

DCALL *name* [,*p1*...[,*p20*]]

This call generates a direct call to the external routine *name*. Note that direct calls do not contain DEFS to the return address. *name* may be in either CODE or DATA space in CDS mode.

PCALL

PCALL *name* [,*p1*...[,*p20*]]

This call is provided for backward compatibility with the old CDSLB macro library. It only calls CDS routines from CDS code. If called in non-CDS mode, it will note the error.

DLCALL

DLCALL *name* [,*call_reg*[,*p1*...[,*p20*]]]

DLCALL is to be used to call local subroutines. *name* is the name of the subroutine and should be defined with a DLENTRY call in the same NAM/END module.

[call reg] is the register to use to call *name* in CDS mode. It must be A, B, or Y. *call_reg* is optional only if this is not the first DLCALL to *name* and *names*'s DLENTRY has not yet been processed. If *call_reg* is optional and is provided, it must match the first definition.

Defs to *p1* to *p20*, if supplied, are generated after the call code. These defs must be accessed with the LOADPARMADD macro since in CDS mode they are not in DATA space.

DLCALL generates a one-word call plus the required parameter DEFS in non-CDS mode. DLCALL generates a two-word call plus the required parameter DEFS in CDS mode.

UCALL

UCALL *name* [,*p1*...[*p20*]]

This macro in all cases generates a:

```
JSB name
DEF RTN
DEF p1
:
DEF pn
```

This is a valid CDS call only if *name* is an RPL entry. That is, *name* is really a machine instruction in CSD code. In non-CDS code, *name* can be a standard subroutine. EXEC in RTE-A and RTE-6/VM satisfies this condition and should be called in this way.

DUCALL

```
DUCALL name [,p1...[,p20]]
```

This macro is the same as UCALL except that the DEF RTN is omitted. This macro should be used only where the routine *name* is RPed in a CDS environment (that is, *name* is a machine instruction.)

Strings and Data

EMPTYSTRING

```
EMPTYSTRING bytesize [,L]
```

This macro builds a FORTRAN string descriptor to an empty string of length *bytesize* bytes. If the current relocation space is not CODE, the result is:

```
name DEC bytesize
     DBL *+1
     BSS (bytesize+1)/2
```

If the current relocation space is CODE, local space is allocated for the description and deferred initialization code is generated. This macro should be called out of line (that is, it is not executable), so the initialization code is deferred until the INITSTRINGS macro is called. In all cases, the string itself is located at *name*+2.

Unless [L] is coded as the second parameter, the initialize code will build a DATA relative address (not a Q relative address). If [L] (the letter L) is coded, the byte address will be Q relative. Note that Q relative byte addresses cannot usefully be passed to external subroutines.

STRING

```
STRING text
```

This macro generates a string descriptor for the string *text* and initializes the string to *text*. If the current code space is not CODE, the macro produces:

```
name DEC :L:text                (length of text in bytes)
     DBL *+1
     ASC (:L:text+1)/2,text
```

If the code space is CODE, local space is allocated, and initialize code is generated by a call to the EMPTYSTRING macro. Deferred code is then generated to initialize the string with *text*. The *text*, if more than four bytes in length, is put in line at the location of the call to STRING. This macro generates in-line data and must not be executed. The deferred code will be put in line on the next call to INITSTRINGS. If *text* is four bytes or less, SBT, STA, or DST instructions, as appropriate for the size, are used and the actual data is obtained by using literals.

The MOVE and COMPARE Macros

The MOVECODETODATA, MOVEWORDS, MOVEBYTES, and COMPARE macros all require three parameters. These are expected to be addresses such that LDA *arg* would get the proper thing to the register. Checks are made to see if A or B or both are passed and the code generated is adjusted to suit. In general, the macros can generate the best code if you let them load the registers. Of course, if the register is already loaded from the prior code sequence, code A or B in the proper area.

MOVECODETODATA

`MOVECODETODATA` *from*`,`*to*`,`*count*

This macro generates in-line code to move *count* words from *from* in CODE space to *to* in DATA space. The *to* address may be in either LOCAL or STATIC space. In non-CDS mode, this macro generates a simple move words. If *count* is a literal (for example, =D10), the macro will avoid the LDX instruction in order to save DATA space. The parameters may be A, B, or any memory reference. For *to* and *from* these should be defs to the areas to move to/from (=L(*to*) is OK) as long as *to* is defined).

count is assumed to be the address of the actual count (that is, =D10 or TEN, where TEN DEC 10 exists somewhere in DATA space). If *count* refers to A or B in a non-CDS environment, a location is allocated, and the register is stored. For moves of 2 words or less, you should use:

```
   DLD =S (or =J)          for two words
   DST
or
   LDA =S (or =D or =B)   for one word
   STA
```

MOVEWORDS

`MOVEWORDS` *from*`,`*to*`,`*count*

This macro generates in-line code to move *count* words from *from* in data space to *to* in data space. In CDS code space, the X-Register is used for the count. In all cases, A and B will end up being *from+count* and *to+count*. In CDS code space, the addresses are adjusted to base relativity (see the MW00 instruction).

MOVEBYTES

`MOVEBYTES` *from*`,`*to*`,`*count*

This macro generates in-line code to move *count* bytes from *from* in data space to *to* in data space. In CDS code space, the X-Register is used for the count. In all cases, A and B will end up being *from+count* and *to+count*. In CDS code space, the addresses are adjusted to base relativity (see the MB00 instruction). Note that *from* and *to* are to be byte addresses. You may code =L(foo+foo) or A or B. If A or B is coded, it is assumed that the address is in the specified register. The macro will move it if it is the wrong register.

COMPAREWORDS

```
    COMPAREWORDS arg1,arg2,count
```

This macro generates in-line code to compare *count* words at *arg1* in data space to *arg2* in data space. In CDS code space, this macro generates a call to a data space routine to do the actual compare. Addresses are adjusted for base relativity and, on exit, are as the CMW instruction leaves them. If *count* is =d1 or =b1, the CPM instruction is used.

Note This macro, in code space, has a high overhead. The execution time is increased as if an additional 15-17 words were compared.

COMPAREBYTES

```
    COMPAREBYTES count [,BASEOK]
```

This macro generates in-line code to compare *count* bytes at A-reg in data space to B-reg in data space. In CDS code space, this macro generates a call to a data space routine to do the actual compare. Addresses are adjusted for base relativity and on exit are as the CBT instruction leaves them. This macro expects A and B to contain the byte addresses of the strings to be compared. Except for adjusting for base relativity, it leaves A and B as the CBT instruction does. In code space, this macro generates code that uses the E-Register.

If the BASEOK parameter is coded as exactly ‘BASEOK’ (lowercase is ok), the macro does not generate code to correct byte addresses that point at the local space. This saves considerable time and also saves the E-Register.

Note This macro, in code space, has a high overhead. The execution time is increased as if an additional 40-41 bytes were compared. If BASEOK is coded, this overhead is cut to 19-20 equivalent bytes.

LOCAL

```
    name LOCAL size
```

In non-CDS mode, this macro builds the deferred line:

```
    name BSS size
```

The deferred lines will be produced in the order given at the next BREAK or END macro call. In CDS mode, if the max number of call parameters is known (that is, it was given in NEWSUB or on ENTRY, or OENTRY has been processed), this macro generates:

```
    name EQU ...+n
```

Where *n* is the next available location in local space. This macro keeps track of *n*, advancing it by *size* on each call. If the maximum number of call parameters is not known, a deferred line is generated as above, starting with *n*=0.

These lines will be generated by the first call to ENTRY, OENTRY, or END, which will affix +*mp* to the end where *mp* is max parameters. <...> is defined by NEWSUB to be the first LOCAL.

BREAK

```
BREAK
```

This macro should be called whenever a place outside of the execution path is reached. In CDS mode, it generates a :OP: BREAK and, in non-CDS mode, it generates a LIT opcode and flushes deferred data declarations from prior LOCAL calls. In non-CDS mode, data labels are defined for pass two by LOCAL calls as needed, followed by a :OP: BREAK call. The LOCAL calls are executable (that is, no code is put in line) while the BREAK macro is not.

RELOC

RELOC *space*[,ALLOC,*name*]

This macro traps all RELOC requests. Since CODE, DATA, and STATIC are only allowed in CDS assemblies, if the current option is CDSOFF, this macro changes the RELOC requests as follows:

```
RELOC CODE    to RELOC PROG
RELOC DATA    to RELOC PROG
RELOC STATIC  to RELOC SAVE
```

In this regard, modules should be written as if they are to run in CDS mode; that is, the standard RELOC commands should be given for CDS mode. The RELOC macro also keeps track of the current space for other macros that need to know the relocation space to generate correct code. In particular, the xCALL, xENTRY, and xSTRING macros need this information.

Note Since in non-CDS, both CODE and DATA are in the same relocation space, you should not attempt to execute through a RELOC DATA.

For example:

```
     lda foo
     sta bar
     reloc data     ; In CDS mode this works fine, but
foo  dec 432        ; in non-CDS mode, this line will be executed
bar  bss 1          ; as will this one.
     reloc code
     sta foobar
      :
      :
```

END

```
END [label]
```

In addition to doing the normal :OP: END operations, this macro completes the ENTRY, OENTRY, and DENTRY number of locals requirement, flushes any deferred locals, and calls BREAK to flush any deferred non-CDS variables. It will indicate an error if *label* is empty, and no ENTRY, OENTRY, or DENTRY has been seen. Also, if NEWSUB declared the number of parameters and no ENTRY, OENTRY, or DENTRY used all of them, an error is indicated.

O

Program Types

This appendix defines all of the program types used with the RTE-6/VM and RTE-A operating systems (Table O-1). Tables O-2 and O-3 describe the ways in which the program types are handled by each operating system.

Table O-1. Program Types

Type	Description
0	System program or driver
1	Memory-resident program
2	Real time program (uses RT or BG partitions)
3	Background disk-resident (uses BG partitions only)
4	Large background disk-resident (uses BG partition only)
5	Segment or overlay (BG or RT, determined by the program main)
6	Extended background (RTE-6/VM), or library routine that becomes part of MR library if referenced by MR program
7	Disk-resident library routine, appended to calling program
8	Deletion of program or routine. If module is main program, it is dropped from generation. If module is a subroutine, RTxGN appends it to programs referencing module, then drops it from system generation. (Essentially an online load)
9	Memory-resident with BG common (reversed common)
10	Real-time with BG common (reversed common)
11	Background with RT common (reversed common)
12	Large background with RT common (reversed common)
13	Table Area II module, recommended for HP use only
14	Type 6 library module,forced into MR library
15	Table Area I module, recommended for HP use only
16	Slow boot (reconfigurator only)
18	Real-time mapped with SSGA access
19	Background mapped with SSGA access
20	Large background mapped with SSGA access
21-24	Not used
25	MR, accessing BG common + SSGA
26	RT, accessing BG common + SSGA
27	BG, accessing RT common + SSGA
28	LB, accessing RT common + SSGA
29	Not used
30	SSGA resident table or module
31-79	Not used (See notes 2 and 3)
519	BLOCK DATA subprogram (Special type)

Table 0-1. Program Types (continued)

Notes:

1. Abbreviations used:
   MR = Memory-Resident
   RT = Real-Time
   LB = Large Background (RTE-6/VM only)
   EB = Extended Background (RTE-6/VM only)
   SSGA = SubSystem Global Area (Named system common)

2. Adding 80 to any executable program type code causes the program to be scheduled automatically at bootup by RTE-6/VM.

   Only one program may be designated with the +80 type code addition. It may only be added in the parameter phase of generation, and not as a compiled program type code.

3. Adding 128 to any executable program type code specifies that the program is not to be renamed by FMGR if the ID segment was created by an RP command.

4. Types 0 through 5 are generally program mains with primary entry points.

Table O-2. Program Type Handling Under RTE-6/VM

Type	Description
0	RT6GN loads into system area. LOADR loads as Type 3. LINK loads as Type 6.
1	If a program main, RT6GN loads into memory-resident area. LOADR forces to Type 3.
2	RT6GN loads in RT disk-resident area. LOADR forces to Type 3. LINK forces to Type 2.
3	RT6GN, LOADR, LINK load into BG disk-resident area.
4	RT6GN and LINK load as per type. LOADR loads as Type 3.
5	RT6GN, LOADR, LINK treat as a segment.
6	(Can be a program main or a subroutine.) If a subroutine called by a memory-resident program, relocated into a memory-resident library during generation. After loading, becomes Type 7. If a program main, LINK and RT6GN treat as Extended BG program.

Table 0-2. Program Type Handling Under RTE-6/VM (continued)

Type	Handling
Note: Types 7 through 519 should be subroutines.	
7	RT6GN can put in system library.
8	If a main, deleted from system during generation. If a subroutine, used to satisfy external references during generation, but not loaded into relocatable library of disk. If a main, LOADR treats as Type 3, LINK treats as Type 6. If not a main, LINK and LOADR treat as Type 7.
9-12	Generated per program type, with accesses as defined.
13	(Pointers and system values defined at generation.) Table Area II is a combination of relocated Type 13 modules and system tables built by generator.
14	(Must NOT be a program main.) After memory-resident loading, becomes Type 7.
15	(System entry points must be in system and user maps.) Table Area I is a combination of these relocated Type 15 modules and I/O tables built by generator.
16-20	Generated per program type, with accesses as defined.
25-28	Generated per program type, with accesses as defined.
30	(Should NOT be program mains.) RT6GN loads in SSGA. LOADR, LINK treat as Type 7.
519	Block Data subprogram. Module contains data only.

Notes:

1. In some cases the primary type code may be expanded by adding 8, 16, 24, or 512 to the number.
2. Refer also to the *RTE-6/VM Programmer's Reference Manual* for more information on program types.

Table O-3. Program Type Handling Under RTE-A

Type	Handling
0-4	No meaning.
5	RTAGN recognizes as a segment and declares it illegal. LINK treats as a segment.
6	RTAGN excludes fixed (relocated) occurrences from generator snap file.
7	RTAGN can put in system library.
8-13	No meaning.
14	(Must NOT be program main.) After memory-resident loading, become Type 7.
15-28	No meaning.
30	(Should NOT be program main.) RTAGN loads in SSGA. LINK treats as Type 7.
519	No meaning.

Index