CRAY RESEARCH, INC.

CRAY COMPUTER SYSTEMS

CAL ASSEMBLER VERSION 2 REFERENCE MANUAL

SR-2003

RECORD OF REVISION

PUBLICATION NUMBER SR-2003

Each time this manual is revised and reprinted, all changes issued against the previous version are incorporated into the new version and the new version is assigned an alphabetic level.

Every page changed by a reprint with revision has the revision level in the lower righthand corner. Changes to part of a page are noted by a change bar in the margin directly opposite the change. A change bar in the margin opposite the page number indicates that the entire page is new. If the manual is rewritten, the revision level changes but the manual does not contain change bars.

Requests for copies of Cray Research, Inc. publications should be directed to the Distribution Center and comments about these publications should be directed to:

CRAY RESEARCH, INC.
2520 Pilot Knob Road
Suite 310
Mendota Heights, Minnesota 55120

Revision	Description
	February 1986 - Original printing.

PREFACE

The CAL Assembler Version 2 allows the user to express symbolically all hardware functions of a mainframe for a Cray Research, Inc. CRAY-2, CRAY X-MP, or CRAY-1 Computer System. This detailed and precise level of programming is of special aid in tailoring programs to the architecture of a Cray mainframe and writing programs requiring code that is optimized to the hardware.

Augmenting the instruction repertoire of CAL is a versatile set of pseudo instructions that provide the user with a variety of options for generating macro instructions, controlling list output, organizing programs, and so on.

Except where indicated, the content of this manual applies to all series of Cray Research, Inc., computers. Detailed information about the Cray operating systems COS and UNICOS† is presented in separate Cray Research, Inc. publications.

The system macro instructions for CRAY X-MP and CRAY-1 Computer Systems that are available with CAL Version 2 are described in the Macros and Opdefs Reference Manual, CRI publication SR-0012.

The system macro instructions for CRAY-2 Computer Systems that is available with CAL Version 2 are described in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013.

---

† UNICOS is derived from the AT&T UNIX system; UNIX is a trademark of AT&T Bell Laboratories.

CONTENTS

5. PSEUDO INSTRUCTIONS (continued)

APPENDIX SECTION

FIGURES

TABLES

DIAGRAMS

INDEX

1. INTRODUCTION

Cray Assembly Language, Version 2 (CAL), is a powerful symbolic language for the generation of object code to be loaded and executed on a Cray Computer System.

CAL source programs consist of sequences of source statements. The source statement can be a symbolic machine instruction, pseudo instruction, a macro instruction, or an opdef instruction. The symbolic machine instructions provide a means of expressing symbolically all functions of a Cray mainframe. Pseudo instructions allow programmer control of the assembly process. Macros and opdefs allow the programmer to define instruction sequences and call them later in the program.

Features inherent in CAL include:

- Free-field source statement format. Size and location of source statement fields are largely controlled by the user.
- Source statements (with some exceptions) can be entered using uppercase, lowercase, or mixed-case letters.
- Control of local and common sections. You can assign code or data segments to specific areas.
- Preloaded data. Data areas can be defined during assembly and loaded with the program.
- Data notation. Data can be designated in integer, floating-point, and character code notation.
- Word and parcel address arithmetic. Addresses can be specified as either word or parcel addresses.
- Listing control. You can control the content of the assembler listing.
- Micro coding. A character string can be defined in a program and substituted for each occurrence of its micro name in the program.
- Macro coding. Sequences of code are defined in a program or on a library, are substituted for each occurrence of the macro name in the program, and use parameters supplied with the macro call.

1.1 EXECUTION OF THE CAL ASSEMBLER

The CAL assembler executes under the control of the Cray operating systems, COS and UNICOS. It has no hardware requirements beyond those required for the minimum system configuration.

The assembler is loaded and begins executing as a result of the CAL invocation statement that is specified by the user. Parameters on the invocation statement specify characteristics of an assembler run, such as the file containing source statements and list output. See section 2 of this publication for descriptions of the CAL control statement used with COS and the CAL command line used with UNICOS.

The source statements can include more than one CAL program segment. The assembler assembles each program segment as it is encountered in the source. The assembler makes two passes for each program segment to be assembled. During the first pass, the assembler reads each source language statement instruction, expands sequences (such as macro instructions), generates the machine function codes, and assigns memory. During the second pass, the assembler substitutes values for symbolic operands and addresses and generates the object code and an associated listing.

The object code must be linked and loaded before execution. References to external symbols are resolved during the link and load phase. The absolute file created by the linker/loader is ready for execution.

1.2 MANUAL ORGANIZATION

This publication is organized as follows:

- Section 2, Operating Systems, describes the CAL invocation statements that execute under the Cray operating systems, COS and UNICOS, program environment and binary definition files.
- Section 3, The CAL Program, describes the organization of a CAL program.
- Section 4, Cray Assembly Language, describes the statement syntax of the CAL program.
- Section 5, Pseudo Instructions, describes the pseudo instructions that are available in CAL.

Appendixes to this publication provide the following information:

- A description of CAL instruction and syntax in Backus-Naur Form (BNF)

- A list of CAL pseudo instructions
- A list of CAL listing messages
- A list of CAL diagnostic messages
- A list of the character sets supported by CAL

Symbolic machine instructions for specific Cray Computer Systems are not included in this manual. For a description of the symbolic machine instructions available with your Cray Computer System, see the Symbolic Machine Instructions Reference Manual, publication SR-0085, and the CRAY-2 Computer System Functional Description, publication HR-2000.

1.3 CONVENTIONS

This publication uses the Backus-Naur Form (BNF). The following general conventions are used in this manual.

italics	Lowercase italicized letters, numbers, or symbols indicate variable information.
underlining	In presenting parameter options, underlining indicates default options.
. . .	A sequence of code is missing from the program.

Throughout this manual, CAL format (syntax) is presented using the following header:

Location	Result	Operand

Throughout this manual, examples of CAL source statements are represented using the following header:

Location	Result	Operand	Comment
1	10	20	35

The following BNF conventions are used in this manual:

x | *y*	Indicates that either *x* or *y* is valid
"x"	Indicates the x is a literal or terminal; the symbol within the quotation marks should be entered exactly as specified. The quotation marks, however, should not be entered.
x	Indicates that x is a nonterminal symbol. A definition of x can be found elsewhere in the syntax representation. For example, the nonterminal symbol octal-digit is defined as follows. octal-digit ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7" . If the nonterminal symbol consists of more than one word, the words are hyphenated.
::=	Follows a nonterminal symbol with a string of symbols that replace the nonterminal symbol
[x]	Indicates x is optional
{x}	Indicates 0 to *n* occurrences of x are valid
.	Indicates the end of a description

2. OPERATING SYSTEMS

CAL Version 2 is a portable assembler that includes the following:

- Multiple operating system interfaces
- Support for binary definition files

2.1 OPERATING SYSTEM INTERFACES

CAL Version 2 interfaces with the following operating systems and Cray Computer Systems:

- Cray Operating System (COS) on CRAY X-MP and CRAY-1 series mainframes
- Cray Operating System (UNICOS) on CRAY-2 and CRAY X-MP and CRAY-1 series mainframes

2.1.1 CRAY OPERATING SYSTEM COS

A typical CAL job on a Cray computer running the Cray operating system COS contains the following files:

- A job control language (JCL) file of COS control statements
- A CAL source file

End-of-file indicators divide files from each other. An end-of-dataset indicator follows the last file. (The actual representation of the end-of-file and end-of-dataset indicators depends on the front-end computer.) Together these files comprise a job dataset, named $IN by COS.

The job dataset is submitted to the Cray computer for processing through a front-end computer. The method of submitting the job depends on the front-end computer.

A job's output dataset (named $OUT by default) is returned to the front-end computer when the job completes. The job's output dataset includes a program listing (by default), any output created by the job, and the job's logfile. The logfile, containing a history of the job and other aspects of running a job on a Cray computer, is described in more detail in the COS Version 1 Reference Manual, publication SR-0011.

CAL generates two kinds of messages during assembly: listing and diagnostic. Listing messages are generated by the assembler when a syntax or semantic error is encountered. A message is printed on the listing beneath every source statement that was flagged by the assembler. A pointer identifies the location in the source statement that corresponds to the message that was issued.

CAL generates five levels of diagnostic messages that are divided into two classes: user information about the assembly (comment, note, and caution) and CAL assembler errors (warning and error). All diagnostic messages are written to the logfile.

Diagnostic user messages are classified by level of severity as follows:

- Comment - Statistical information
- Note - Possible assembly problems
- Caution - Definite user errors during assembly of the program

These messages are always printed in the logfile and include messages about segments processed, time of assembly, number of messages, and so on. These messages are numbered 1 through 99 and are listed in appendix D, Diagnostic Messages.

CAL diagnostic assembler messages (warning and error) are printed only if the assembler is malfunctioning. Therefore, it is unlikely that they will ever appear in your job's logfile. If a diagnostic message with a priority of warning or error message ever does appear in your job's logfile, contact your local site analyst. These messages are numbered 100 and greater and are not listed in this manual.

2.1.1.1 JCL file

A simple CAL job may contain the following COS control statements in its JCL file followed by an end-of-file marker, the CAL source, and another end-of-file marker:

```
JOB,JN=TEST.
ACCOUNT,AC=... .
CAL.
SEGLDR.
$ABD.
/EOF
         IDENT       TEST
         .
         .
         .
         END
/EOF
```

The JOB statement is a required statement that defines the job to COS. At the minimum, it must contain a JN parameter to assign the job name.

The ACCOUNT control statement presents the user's account number, which may be required by a site before access is granted to the system.

The CAL statement causes the CAL assembler to be loaded and executed. CAL control statement parameters are described in the following subsection.

The SEGLDR statement links and loads the assembled program.

The $ABD statement executes the assembled program.

These and other control statements are described in the COS Version 1 Reference Manual, publication SR-0011.

2.1.1.2 CAL control statement

The COS NEWCAL control statement invokes the CAL Version 2 assembler. The user selects assembler parameters either explicitly by listing them on the control statement or implicitly by accepting the default values. All parameters have default values. Parameters are order independent and are optional.

Format:

```
NEWCAL,I=[idn{:[idn]}],L=ldn,E=edn,B=bdn,X=xdn,S=[sdn{:[sdn]}],
  T=tdn,SYM=symdn,ALLSYMS,ABORT,CPU=[primary]{:[charac]},NLIST,
  LIST=[name{:[name]}],options,ML=level,MC=count,FORMAT=format,
  EDIT=edit.
```

I=[*idn*{:[*idn*]}]
: Name of dataset containing source statement input. The default is $IN. CAL reads source statements from dataset *idn* until an end-of-file is encountered. One or more dataset names can be entered and are processed as if appended.

L=*ldn*
: Name of dataset into which list output is written. The default is $OUT. CAL writes one file of output. If L=0, no listing is written.

E=*edn* Name of dataset on which messages are written. The default is no message dataset if the list output is on $OUT, otherwise, the default is $OUT. CAL writes source statements containing messages to this dataset as one file.

Specifying E causes a message dataset to be generated on a file named $OUT. If the message dataset name, *edn*, is the same as the listing dataset name, list output is written. If E=0, the message dataset is not listed.

B=*bdn* Name of dataset to receive binary load data. The default is $BLD. CAL writes binary load data to this dataset, one record per program module. An end-of-file is not written. If B=0, no binary load data is written.

X=*xdn* Binary symbol table for the global cross-reference generator, SYSREF. Each record contains cross-reference information for the global symbols in one particular program unit. The default, equivalent to specifying X=0, writes no global cross-reference records. If X is specified and the listing is suppressed (L=0), the cross-reference file is not created. If X is specified without a value, the information is written to $XRF.

S=[*sdn*{:[*sdn*]}]
Binary definition dataset name. The default is $SYSDEF. If S=0 is specified, no binary definition datasets are used. *sdn* can be a single dataset name or a list of dataset names separated by colons. The following is an example of specifying a list of dataset names:

```
S=$SYSDEF:OURDEF:MYDEF
```

Binary definition datasets are processed in the order in which they are specified.

T=*tdn* Binary definition. Specifies dataset name to which all global macros, opdefs, symbols, micros, and OPSYN assignments are written. The default, equivalent to specifying T=0, is that no binary definition dataset is written. If T is specified without a value, the binary dataset is written to $BDF.

SYM=*symdn* Name of dataset where the optional symbol table is to be written. The default is that no symbol table dataset is generated by CAL. If SYM is specified without a value, the symbol text is written to the same dataset as the binary load data.

ALLSYMS Forces a symbol table to be generated with all symbols; normally nonreferenced symbols are not included. If the SYM option is not specified on the CAL control statement, the symbol text is written to the same dataset as the binary load data.

ABORT Abort mode. If this parameter is present and if any diagnostic messages of priority caution, warning, or error were issued to the logfile, CAL aborts the job after the assembly of the program.

If this parameter is omitted or if diagnostic error messages of priority caution, warning, or error were not encountered, CAL exits normally and job processing continues the next control statement in the job.

CPU=[*primary*]{:[*charac*]}

Cray computer to execute CAL source code. The default is that code is generated for the characteristics of the machine specified in the TARGET control statement. If there is no previous TARGET control statement in the JCL stream, code is generated for the characteristics of the host machine. For more information about the TARGET control statement, see subsection 2.1.1.3, The COS Environment.

If the CPU option instruction set looks like this,

CPU=:*charac*{:[*charac*]}

where the *primary* is not specified and one or more *charac* are given, the *primary* stated on the TARGET control statement is used. Any *charac* that are not specified are taken from the TARGET control statement.

If the CPU option instruction set looks like this,

CPU=*primary*{:[*charac*]}

where the *primary* is specified and the *charac* may or may not be given, the specified *primary* overrides that of the TARGET control statement. Any *charac* that are not specified are taken from the defaults for the given *primary*.

primary The type of Cray computer. The *primary* options may differ from site to site. The commonly used options are:

*HOST The machine on which the assembler is currently running

*TARGET	The machine that is specified in the TARGET control statement
CRAY-X4	CRAY X-MP Models 48 and 416
CRAY-X2	CRAY X-MP Models 22, 24, and 28
CRAY-X1	CRAY X-MP Models 11, 12, 14, and 18
CRAY-XMP	CRAY X-MP
CRAY-1M	CRAY-1M
CRAY-1S	CRAY-1S
CRAY-1B	CRAY-1B
CRAY-1A	CRAY-1A
CRAY-1	CRAY-1
CRAY-2	CRAY-2

charac The features of the *primary* computer.

The CRAY-2 series has no special feature options.

The CRAY X-MP and CRAY-1 Computer Systems permit you to specify the following logical and numeric traits:

Logical Traits	Description
AVL	Additional vector logical
NOAVL	No additional vector logical
BDM	Bidirectional memory
NOBDM	No bidirectional memory
CIGS	Compressed index and gather/scatter
NOCIGS	No compressed index and gather/scatter
CORI	Control operand range interrupts
NOCORI	No control operand range interrupts
EMA	Extended memory addressing
NOEMA	No extended memory addressing
HPM	Hardware performance monitor
NOHPM	No hardware performance monitor
PC	Programmable clock
NOPC	No programmable clock
READVL	Read vector length
NOREADVL	Do not read vector length
STATRG	Status register
NOSTATRG	No status register
VPOP	Vector pop count
NOVPOP	No vector pop count
VRECUR	Vector recursion

Numeric Traits	Description
NOVRECUR	No vector recursion
BANKBUSY=*n*	Bank busy time in clock periods†
BANKS=*n*	Number of memory banks†
CLOCKTIM=*n*	Clock time in picoseconds†
IBUFSIZE=*n*	Instruction buffer size in words†
MEMSIZE=*n[c]*	Memory size in words; *c* can be one of the following:† K = $n*1000_8$ words M = $n*1000000_8$ words
MEMSPEED=*n*	Memory speed in clock periods
NUMCLSTR=*n*	Number of cluster registers
NUMCPUS=*n*	Number of cpus†

NLIST Ignores all LIST pseudos in the code including those specified by LIST in the control statement

LIST=[*name*{:[*name*]}]
Name of LIST pseudo instructions to be processed. A LIST pseudo instruction with a matching location field name is not ignored. A LIST pseudo with a location field name that does not match a name specified on the CAL control statement is ignored.

A *name* can be a single name or can be a list of names separated by colons, for example:

```
LIST=TASK1:TASK2:TASK7
```

If just LIST is specified, all LIST pseudo instructions are processed, regardless of the location field name.

options Listing control options. Any of the following listing control options can be specified to enable or disable a listing feature. The selection of an option on the CAL control statement overrides the enabling or disabling of the corresponding feature on a LIST pseudo instruction. Refer to the description of the LIST pseudo in subsection 5.6, Listing Control, for more details about these options.

† *n* represents an unsigned decimal number.

Defaults are underlined.

ON	Enables source statement listing
OFF	Disables source statement listing
ED	Enables listing of edited statements
NED	Disables listing of edited statements
XRF	Enables cross-reference
NXRF	Disables cross-reference
XNS	Includes nonreferenced symbols in cross-reference
NXNS	Does not include nonreferenced symbols in cross-reference
LIS	Enables listing of listing control pseudo instructions
NLIS	Disables listing of listing control pseudo instructions
TXT	Enables global text source listing
NTXT	Disables global text source listing
MAC	Enables listing of macro and opdef expansions
NMAC	Disables listing of macro and opdef expansions
MBO	Enables macro binary only
NMBO	Disables macro binary only
MIC	Enables listing of generated statements before editing within an expansion
NMIC	Disables listing of generated statements before editing within an expansion
MIF	Enables macro conditional listing
NMIF	Disables macro conditional listing
DUP	Enables listing of duplicated statements
NDUP	Disables listing of duplicated statements

ML=*level* Priority of listing messages received on output and message listing. *level* can be: COMMENT, NOTE, CAUTION, WARNING, or ERROR; the default is WARNING. Specific levels are described under the MLEVEL pseudo instruction. Message descriptions are in appendix D, Diagnostic Messages.

level indicates the threshold for listing messages; COMMENT is considered to be the lowest level and ERROR to be the highest level. When a threshold level is specified, the specified level and all levels above it are printed.

For example, if ML=CAUTION, CAL prints caution, warning, and error messages. If ML=ERROR, CAL prints only error messages. If ML=COMMENT, CAL prints all message levels (comment, note, caution, warning, and error).

When ML is set on the control statement, the MLEVEL pseudo is ignored. In other words, the ML specification cannot be overridden by the MLEVEL pseudo.

MC=*count* Message count. Specifies how many messages print on the listing. For example, if *count* is set to 200, the line of code that contains approximately the 200th listing message causes display of a message saying the maximum number of messages have been encountered and no more messages are listed. The default for MC is 100.

FORMAT=*format*

FORMAT options are OLD and NEW. FORMAT sets the statement format to old (CAL Version 1 style) or new. If FORMAT is not specified, the default is new for CRAY-2 Computer Systems and old for CRAY X-MP and CRAY-1 Computer Systems.

Statement format can be modified by the FORMAT pseudo within an assembler program, but the default established by the FORMAT option on the CAL invocation statement is reactivated at the beginning of each segment. For more information about the FORMAT pseudo, see subsection 5.3, Mode Control.

EDIT=*edit* Edit options are ON and OFF; the default is ON. EDIT turns the actual editing of statements (concatenation and micro substitution) on and off. If the default is used, editing can be modified by the EDIT pseudo.

Statement editing can be modified by the EDIT pseudo within an assembler program, but the default established by the EDIT assembler option is reactivated at the beginning of each segment. For more information about the EDIT pseudo, see section 5.3, Mode Control, of this publication.

2.1.1.3 The COS environment

The COS environment can be modified using the OPTION and TARGET control statements. NEWCAL uses values set by OPTION and TARGET to establish values for its environment.

LPP parameter on the OPTION control statement - The LPP parameter on the OPTION control statement sets the number of lines per page for output listings. By default, the number of lines per page is 55. The format of the OPTION control statement is as follows:

OPTION,LPP=*n*,....

n Specifies the page length that CAL uses for output listings. *n* is a decimal number that must be a value in a valid range (0-255 for COS and 4 through 999 for CAL); the default is 55. If *n* is outside of the permitted range by CAL (4-255), CAL uses the default value of 55.

For more information about the OPTION control statement, see the COS Version 1 Reference Manual, publication SR-0011.

TARGET control statement - The TARGET control statement identifies the kind of cpu for which CAL is targeting code. The format of the TARGET control statement is as follows:

TARGET,CPU=[*primary*]{:[*charac*]}

primary Machine type. See the CPU parameter for the CAL control statement elsewhere in this section for more information.

charac Characteristics for the target machine. See the CPU parameter for the CAL control statement elsewhere in this section for more information.

For more information about the TARGET control statement, see COS Version 1 Reference Manual, publication SR-0011.

For every COS job, there are two environment descriptions. One, called the host, describes the machine on which the job is currently executing. The other, called the target, describes the machine that language processors may use to determine the machine for which they generate code.

Initially, the host and the target are identical. For the duration of the job, the host remains unchanged. However, the target may be modified by the TARGET control statement at any time and as frequently as needed.

The assembler uses the description of the target if the CPU parameter is not specified on the NEWCAL control statement or if a *primary* is not specified on the CPU parameter. All options specified on the CPU parameter override the current target description during the assembler's execution.

2.1.2 CRAY OPERATING SYSTEM (UNICOS)

A typical interactive session in which a CAL program is assembled on a Cray Computer System that is running UNICOS contains a CAL source file that is assembled, loaded, and executed with a series of commands entered at the keyboard.

CAL does not use the standard input file or standard output file during assembly, but does use the standard error file to report diagnostic messages and source line messages.

CAL generates two kinds of messages during assembly: listing and diagnostic. If the -l and -L options are specified on the CAL as command line, listing messages are generated by the assembler when a syntax or semantic error is encountered. A message is printed on the listing beneath every source statement that was flagged by the assembler. A pointer identifies the location in the source statement that corresponds to the message that was issued. This type of message is also issued to the standard error file.

CAL generates five levels of diagnostic messages that are divided into two classes: user information about the assembly (comment, note, and caution) and CAL assembler errors (warning and error). All diagnostic messages are written to standard error.

Diagnostic user messages are classified by level of severity as follows:

- Comment - Statistical information
- Note - Possible assembly problems
- Caution - Definite user errors during assembly of the program

The -V option must be specified in order for messages with a priority of comment to be printed to standard error. Messages with a priority of note and caution are printed to standard error even if -V is not specified. Messages with a priority of comment, note, and caution are numbered 1 through 99 and are listed in appendix D, Diagnostic Messages.

CAL diagnostic assembler messages with priorities of warning and error are printed only if the assembler is malfunctioning. Therefore, it is unlikely that they will ever appear. If a diagnostic message with a priority of warning or error ever does appear in standard error, contact your local site analyst. These messages are numbered 100 and greater and are not listed in this manual.

2.1.2.1 Interactive assembly

A CAL program can be assembled and executed interactively by entering the following commands at the keyboard:

```
as myfile.s
ld myfile.o
a.out
```

The as command assembles file *myfile.s* and creates file *myfile.o*.

The ld command links and loads the assembled program found in *myfile.o* and creates the executable file a.out.

The a.out command executes the executable file a.out

These and other commands are described in the UNICOS Commands Reference Manual, publication SR-2011.

2.1.2.2 as - CAL command line

The UNICOS as - common Cray assembler (CAL) command line invokes the CAL Version 2 assembler.

Format:

```
as [-o objfile] [-l lstfile] [-L msgfile] [-b bdflist] [-B]
   [-c bdfile] [-g symfile] [-G] [-C cpu] [-h] [-H]
   [-i nlist] [-I options] [-m mlevel] [-n number]
   [-f] [-F] [-j] [-J] [-V] filename
```

The as command assembles the named file. The following options, each a separate argument, can appear in any order, but must precede the filename argument.

-o *objfile*
: Relocatable assembly output; stored in file *objfile*. By default, the relocatable output file name is formed by removing the path name and the .s suffix, if they exist, from the input file and by appending a .o suffix. *objfile* must be processed by a link editor or loader.

-l *lstfile*

Assembly output source listing; stored in file *lstfile*. By default, the output source listing is suppressed.

-L *msgfile*

Assembly output source message listing; stored in file *msgfile*. By default, the output message listing is suppressed.

-b *bdflist*

Reads the binary definition files stored in one or more files. The files named in *bdflist* can be designated using one of the following forms:

- List of files separated from one another by a comma.
- List of files enclosed in double quotes and separated from one another by a comma and/or one or more spaces.

Reads the default binary assembler definitions found in file /lib/asdef unless suppressed with the -B option. The remaining files listed in *bdflist* are read in the order in which they are specified.

-B Suppresses /lib/asdef as the default binary assembler definition file.

-c *bdfile*

Creates the binary definition file *bdfile*. By default, the creation of a binary definition file is suppressed.

-g *symfile*

Assembly output symbol file; stored in *symfile*. *symfile* is used by the system debuggers. By default, the output symbol file is suppressed.

-G Forces all symbols to *symfile* if the -g option is used. Normally, nonreferenced symbols are not included.

-C *cpu* Code is generated for the specified cpu. The default is that code is generated for the characteristics of the host machine. *cpu* has the following syntax:

cpu ::= *primary*{","[*charac*]}
or
cpu ::= ","[*charac*]{","[*charac*]}

primary *primary* can be one of the following Cray Computer Systems:

cray-2	CRAY-2
cray-x4	CRAY X-MP Models 48 and 416
cray-x2	CRAY X-MP Models 22, 24, and 28
cray-x1	CRAY X-MP Models 11, 12, 14, and 18
cray-xmp	CRAY X-MP
cray-1m	CRAY-1 M
cray-1s	CRAY-1 S
cray-1b	CRAY-1 B
cray-1a	CRAY-1 A
cray-1	CRAY-1

charac The features of the *primary* computer.

CRAY-2 Computer Systems have no special options.

The CRAY X-MP and CRAY-1 Computer Systems permit you to specify the following logical and numeric traits:

Logical Traits	Description
avl	Additional vector logical
noavl	No additional vector logical
bdm	Bidirectional memory
nobdm	No bidirectional memory
cigs	Compressed index and gather/scatter
nocigs	No compressed index and gather/scatter
cori	Control operand range interrupts
nocori	No control operand range interrupts
ema	Extended memory addressing
noema	No extended memory addressing
hpm	Hardware performance monitor
nohpm	No hardware performance monitor
pc	Programmable clock
nopc	No programmable clock
readvl	Read vector length
noreadvl	Do not read vector length
statrg	Status register
nostatrg	No status register
vpop	Vector pop count
novpop	No vector pop count
vrecur	Vector recursion

Numeric Traits	Description
novrecur	No vector recursion
bankbusy=*n*	Bank busy time in clock periods†
banks=*n*	Number of memory banks†
clocktim=*n*	Clock time in picoseconds†
ibufsize=*n*	Instruction buffer size in words†
memsize=*n*	Memory size in words†
memspeed=*n*	Memory speed in clock periods†
numclstr=*n*	Number of cluster registers
numcpus=*n*	Number of cpus†

-h Enables all list pseudos regardless of the location field name.

-H Disables all list pseudos regardless of the location field name.

-i *nlist* Restricts list pseudo processing to those pseudos whose location field names are given in *nlist*. The names specified by *nlist* can take one of the following forms:

- List of names separated from one another by a comma
- List of names enclosed in double quotes and separated from one another by a comma and/or one or more spaces.

-I *options*

List options. A list of more than one option must be specified without intervening blanks. It is not permitted to specify conflicting options (the same character in uppercase and lowercase) in the same -I list. *options* can be any of the following:

s Enable source statement listing (default)
S Disable source statement listing

e Enable edited statement listing (default)
E Disable edited statement listing

t Enable text source statement listing
T Disable text source statement listing (default)

† *n* represents an unsigned decimal number.

l Enable listing control pseudo instructions
L Disable listing control pseudo instructions (default)

m Enable macro/opdef expansions binary only
M Disable macro/opdef expansions binary only (default)

d Enable dup/echo expansion
D Disable dup/echo expansion (default)

b Enable macro/opdef/dup/echo expansion binary only
B Disable macro/opdef/dup/echo expansion binary only (default)

c Enable macro/opdef/dup/echo expansion conditionals
C Disable macro/opdef/dup/echo expansion conditionals (default)

p Enable macro/opdef/dup/echo expansion of pre-edited lines
P Disable macro/opdef/dup/echo expansion of pre-edited lines (default)

x Enable cross-reference listing (default)
X Disable cross-reference listing

n Enable nonreferenced local symbols included in the cross-reference (default)
N Disable nonreferenced local symbols included in the cross-reference

-m *mlevel*

Priority for the output listing, the message listing, and the standard error file. *mlevel* can be one of the following:

comment, note, caution, warning, or error

If the -m option is specified, it overrides all MLEVEL pseudo instructions. By default, the priority is warning, and the MLEVEL pseudo instruction controls the message level during assembly.

-n *number*

Maximum number of messages to be inserted into the output listing, the message listing, and the standard error file. *number* must be zero or greater; the default is 100.

-f Enables the new statement format. By default, the old format is used when targeting for a CRAY X-MP or CRAY-1 Computer System; otherwise, the new format is used. Statement format reverts to the format specified on the invocation statement at the end of every assembler segment.

-F Disables the new statement format. By default, the old format is used when targeting for a CRAY X-MP or CRAY-1 Computer System; otherwise, the new format is used. Statement format reverts to the format specified on the invocation statement at the end of every assembler segment.

-j Enables editing; the default is enabled. Editing status reverts to the status specified on the invocation statement at the end of every assembler segment.

-J Disables editing; the default is enabled. Editing status reverts to the status specified on the invocation statement at the end of every assembler segment.

-V Causes the version number of the assembler being run and other statistical information (diagnostic messages of priority, comment, note, and caution) to be written to the standard error file.

filename File to be assembled; all options must precede the file name argument.

2.1.2.3 The UNICOS environment

The CAL assembler is affected by the LPP shell variable from the UNICOS environment. The LPP shell variable sets the number of lines per page for output listings (page length). By default, the number of lines per page is 55.

The UNICOS environment is set as follows:†

```
LPP=n as filenamex.s
```

or

```
LPP=n
as filenamea.s
as filenameb.s
.
.
.
as filenamez.s
```

n Specifies the page length used for output listings. *n* is a decimal number. CAL requires a value in a valid range (4 through 999); the default is 55. If *n* is outside of the valid range, CAL uses the default to set the page length.

† The environment is dependent on the type of shell being used.

filenamea, filenameb...filenamez
Names of the UNICOS files that are being assembled.

If the LPP shell variable is specified before and on the same line as the as command line, the number of lines per page assigned by the LPP shell variable affects only that particular as instruction.

If the LPP shell variable is specified as a separate entry, all of the assemblies that follow use the page length specified by that LPP shell variable for output and message listings.

In the following example, the number of lines per page for the output listings for srca.s and srcb.s is 45:

```
LPP=45
as srca.s
as srcb.s
```

In the following example, the page length for srcd.s is 45. The page length for srce.s, however, reverts to 64. 64 is used because the second LPP shell variable is associated only with file srcd.s:

```
LPP=64
LPP=45 as srcd.5
as srce.s
```

Table 2-1 compares the control parameters for the Cray operating systems COS and UNICOS.

Table 2-1. Comparison of COS and UNICOS Parameters

COS	UNICOS	Comments
I=[*idn*{:[*idn*]}] Default is $IN	*filename* No default	Input source
L=*ldn* Default is $OUT	-l *lstfile* Default is no *lstfile*	Source listing

Table 2-1. Comparison of COS and UNICOS Parameters (continued)

COS	UNICOS	Comments
E=*edn* Default is $OUT	-L *msgfile* Default is no *msgfile*	Message listing
B=*bdn* Default is $BLD	-o *objfile* Default is file name with .s suffix replaced by .o suffix	Relocatable or object file
B=O	Not available	
X=*xdn*	Not available	Cross-reference symbol table
S=[*sdn*{:[*sdn*]}] Default is $SYSDEF	-b *bdflist* Default is /lib/asdef	System definitions or binary definition files
S=O	-B	Suppress the use of binary definition file
T=*bdf*	-c *bdfile*	Create binary definition file
SYM=*sym* ALLSYMS	-g *symfile* -G	Symbol table file Force all symbols to symbol table file
ABORT	Not applicable	Abort mode; when diagnostic messages are sent to the logfile, CAL aborts the job
CPU=*cpu*	-C *cpu*	Target machine
LIST	-h	Enable all list pseudo instructions
NLIST	-H	Disable all list pseudo instructions
LIST=*names*	-i *nlist*	Enable all list pseduos with a matching location field name
options	-I *options*	List pseudo options

Table 2-1. Comparison of COS and UNICOS Parameters (continued)

COS	UNICOS	Comments
ML=*mlevel*	-m *mlevel*	Message level
MC=*count*	-n *number*	Number of messages allowed in the listing
FORMAT=NEW	-f	Enable new format
FORMAT=OLD	-F	Enable old format
EDIT=ON	-j	Enable editing
EDIT=OFF	-J	Disable editing
Not applicable	-V	Causes the version number of the assembler being run and other statistical information to be written

2.2 BINARY DEFINITION FILES

The CAL Version 2 Assembler allows your assembler source program access to previously assembled lines or sequences of code. These preassembled sequences are stored in files that are called binary definition files. These files are analagous to libraries. Binary definition files can be classified in two groups:

- System defined
- User defined

The system-defined binary definition files for the Cray Operating Systems are $SYSDEF for COS and /lib/asdef for UNICOS. These system-defined binary definition files are accessed automatically by CAL unless the assembler is directed otherwise. Binary definition files contain symbols, macros, opdefs, opsyns, and micros that are commonly used by CAL users. For the macros and opdefs available under COS and UNICOS, refer to the Macros and Opdefs Reference Manual, CRI publication SR-0012 and the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013, respectively.

NOTE

System- and user-defined binary definition files are identical in all respects. User-defined binary definition files are created and used the same way that system-defined binary definition files are created and used. They have been treated as separate entities in this discussion in order to encourage you to define binary definition files that meet your particular programming requirements.

User-defined binary definition files can be created by copying the system-defined binary definition files and by editing them in one of the following ways:

- Adding to the system-defined binary definition file
- Redefining a definition in the system-defined binary definition file

or by disabling the recognition of system-defined binary definition files and accumulating the defined sequences entirely from an assembler source program.

You can specify more than one binary definition file with each assembly. If more than one binary definition file is specified, the files are processed from left to right in the order in which they are specified on the S (COS control statement) and -b parameters (UNICOS as command line).

Binary definition files are important, because once lines or sequences of code are assembled and stored in a binary definition file, they can be accessed without being reassembled. The direct access of a binary results in considerable savings in assembler time.

System- and user-defined binary definition fields are defined, created, and used in the same manner.

2.2.1 DEFINING A BINARY DEFINITION FILE

Only certain types of lines or sequences of code are permitted in a binary definition file. Binary definition files are always created from the global part of program seqments and currently accesssed binary definition files, if any. Ordinarily, binary definition files are created from source programs that include a global part in a single segment that does not include a program module.

You can also make additions to binary definition files using assembler source programs that may or may not include program modules. Under no circumstance is any line or sequence of code added to a binary definition file from an assembler program module. Although all additions to binary definition files come from the global part of the segment, not all lines or sequences of code in the global part are added when a new binary definition file is created.

Binary definition files are made up of lines or sequences of code that can be classified as follows:

- Symbols
- Macros
- Opdefs
- Opsyns
- Micros

Every line or sequence of code must fall into one of the classes listed above and satisfy the requirements for that particular class before they are added to a binary definition file.

2.2.1.1 Symbols

CAL accumulates symbols to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of program segments that fit the following requirements:

- Symbols that are to be included in binary definition files cannot be redefinable. To be included in a binary definition file, a symbol must be defined with the = pseudo instruction. Symbols defined with the SET or MICSIZE pseudo instruction are redefinable and therefore are not included in a binary definition file.

- Symbols that are to be included in binary definition files cannot be preceded by %%. This exclusion applies to symbols that are created by the LOCAL and = pseudo instructions.

CAL identifies all of the symbols in the global part of program segments that meet the requirements described above and includes them when a binary definition file is created. In figure 2-1, SYM1, SYM3, and SYM4 meet the requirements and are included. SYM2 (defined in the module), SYM5 (redefinable), %%SYM6 (begins with %%) do not meet the requirements and are not included when a binary definition file is created.

2.2.1.2 Macros

CAL accumulates macros to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within a source program.

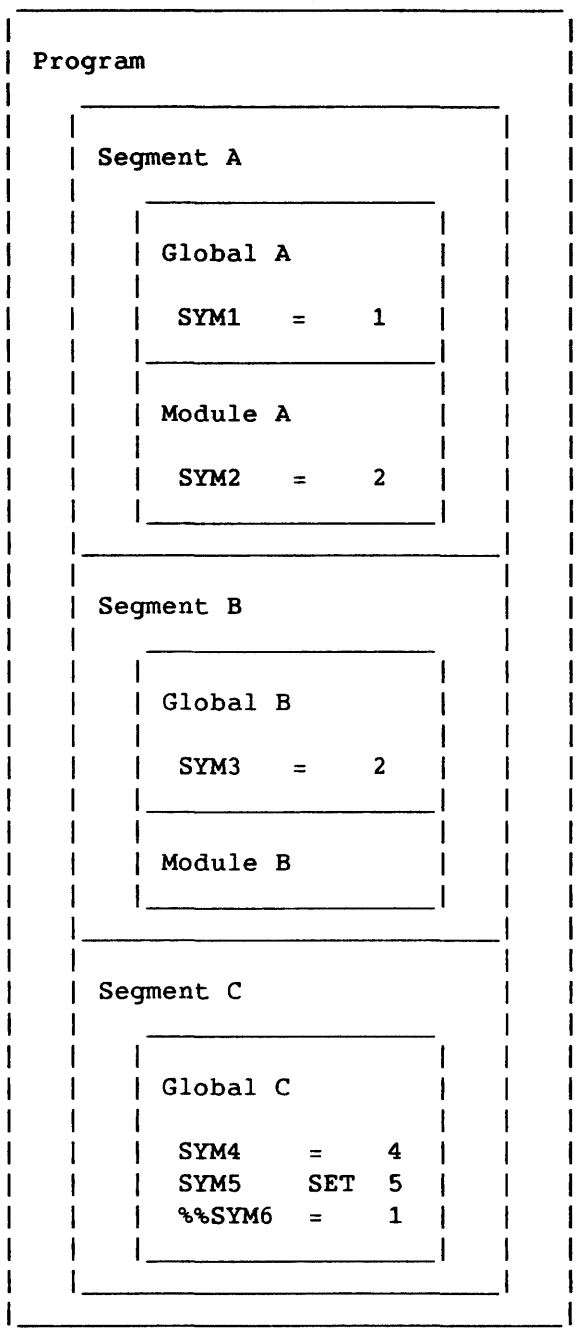

Figure 2-1. Symbols to be Included in a Binary Definition File

2.2.1.3 Opdefs

CAL accumulates opdefs to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within a source program.

2.2.1.4 Opsyns

CAL accumulates opsyns to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within a source program.

2.2.1.5 Micro

CAL accumulates micros to be included in a new binary definition file from all currently accessed binary definition files and from all of the global parts of segments within source program. Only micros that cannot be redefined are included in a binary definition file. If a micro defined by the micro pseudo instruction is to be included into a binary definition file, it must be defined using the CMICRO pseudo instruction.

2.2.2 CREATING BINARY DEFINITIONS FILES

Binary definition files can be created for use with the Cray operating systems COS and UNICOS.

2.2.2.1 Creating new binary definition files for COS

A binary definition file is created under COS when the S and T parameters are included on the COS control statement. The S parameter can be followed by one or more dataset names that are separated by colons.

In the following example, $SYSDEF, OURDEF, and MYDEF are the names of the predefined binary definition files that are to be included along with any symbols, macros, opdef, opsyn, and micros from the global parts of the program segments from the current source program being assembled. The new binary definition file named NEWDEF is defined by the T parameter.

```
NEWCAL,S=$SYSDEF:OURDEF:MYDEF,T=NEWDEF.
```

NEWDEF, the binary definition file created by the T parameter in the previous example, contains the symbols, macros, opdefs, opsyns, and micros that were accumulated by CAL from $SYSDEF, OURDEF, MYDEF, and $IN (default input file). When the next source program is assembled, the following statement would make NEWDEF available as the only binary definition file:

```
NEWCAL,S=NEWDEF.
```

If only the symbols, macros, opdefs, opsyns, and micros accumulated from the global parts of the program segments from the current source program being assembled are to be entered into a binary definition file, the following would be entered:

```
NEWCAL,S=0,T=NEWDEF.
```

The S=0 specification suppresses the default system-defined binary definition file ($SYSDEF) and excludes the definitions in $SYSDEF from the binary definition file being created. If a binary definition file was defined as shown above, the new binary definition file (NEWDEF) would be created and on subsequent assemblies could be specified as follows:

```
NEWCAL,S=$SYSDEF:NEWDEF.
```

In the previous example, two binary definition files ($SYSDEF the system-defined file and NEWDEF a user-defined file) are specified.

2.2.2.2 Creating new binary definition files for UNICOS

A binary definition file is created under UNICOS when the -b and -c parameters are included on the as command line. The -b parameter can be followed by a list of files that are separated by commas or a list of files enclosed in double quotes and separated by spaces or commas.

In the following example, the default system-defined binary definition file /lib/asdef and user-defined binary definition files ourdeffile, and mydeffile are to be included along with the accumulated symbols, macros, opdefs, opsyns, and micros from the global parts of the program segments from the current source program (prog.s) being assembled. The new binary definition file named mynewfile is defined by the -c parameter and is created by CAL.

```
as -b ourdeffile,mydeffile, -c mynewfile prog.s
```

By default, the default binary definition file (/lib/asdef) is always available unless it is suppressed with the -B parameter. If not suppressed, /lib/asdef is the first binary definition file that is read.

Any other binary definition files that are specified by the -b parameter are processed in the order in which they are specified. When the next source program is assembled, the following statement makes mynewfile available as the only binary definition file:

```
as -B -b mynewfile prog.s
```

If only the accumulated symbols, macros, opdefs, opsyns, and micros from the global parts of the program segments from the current source program being assembled are to be entered into a binary definition file, the following could be entered:

```
as -B -c mynewfile prog.s
```

The -B parameter disables the default system-defined binary definition file /lib/asdef and only the accumulated symbols, macros, opdefs, opysns, and micros from the global parts of the program segments from the current source program being assembled are included in the new binary definition file. If a binary definition file was created using the parameters shown above, binary definition file mynewfile could be specified on a subsequent assembly as follows:

```
as -b mynewfile prog.s
```

In the previous example, two binary definition files (/lib/asdef the default system-defined file and mynewfile a user-defined file) are used.

2.2.3 USING BINARY DEFINITION FILES

Binary definition files allow users to access lines or sequences of code that have been previously assembled. Binary definition files are accessed using the S parameter on the COS control statement and the -b parameter on the UNICOS as command line. The following checks are run on binary definition files when they are accessed:

- Compatibility checking
- Multiple references to the same definition

2.2.3.1 Compatibility checking

CAL allows the user to access any previously defined file with one restriction. Binary definition files are marked with the kind of cpu (CRAY-2, CRAY X-MP, or CRAY-1 Computer System) for which they were created. If a binary definition file is created on a CRAY X-MP or CRAY-1

Computer System and is specified for a CRAY-2 Computer System, or if a binary definition file is created on a CRAY-2 Computer System and is specified for a CRAY X-MP or CRAY-1 Computer System, the binary definition file is not accepted and the following message is issued:

```
Incompatible version of binary definition file 'file'
```

This check ensures that the machine on which the binary definition file was created is compatible with the program that is attempting to use it. Some CAL Version 2 pseudo instructions have restricted use that is based on hardware and software requirements. The binary definition file compatibility check protects users from getting binary definition files mixed and ensures that hardware and software restrictions are not violated.

2.2.3.2 Multiple references to a definition

CAL checks for multiple references to definition names (functional names for macros and opsyns, location field names for symbols, and micros, and syntax for opdefs. CAL handles multiple references to definitions with the same functional name, location field name, or opdef syntax as follows:

Symbols - If a symbol is defined in more than one binary definition file, the definitions are compared. If the definitions are identical, CAL disregards the duplicates and makes one entry for the symbol from the binary definition files. If a symbol is defined more than once and the definitions are not identical, CAL uses the last definition associated with the location field name and issues the following diagnostic message:

Symbol '*name*' is redefined in file '*file*'

Macros - If a macro with the same functional name is defined in more than one binary definition file, the definitions are compared. If the definitions associated with the macro's functional name are identical character by character, CAL disregards the duplicate definition and makes one entry for the macro from the binary definition files. If the macro's functional name is used more than once, and the definitions associated with the functional name are not identical character by character, CAL uses the definition associated with the last reference to the functional name and issues the following diagnostic message:

Macro '*name*' in file '*file*' replaces previous definition

If a macro is defined with the same functional name as a pseudo instruction, the macro replaces the pseudo instruction and CAL issues the same message as shown above.

Opdefs - If an opdef with the same syntax is defined in more than one binary definition file, the definitions of the opdefs are compared. If the definitions of the two opdefs are exactly the same, CAL disregards the duplicate definition and makes one entry for the opdef from the binary definition files. If the same syntax appears more than once and the definitions are not exactly the same, the syntax associated with the last reference to the opdef is used as its definition. CAL also issues the following diagnostic message:

Opdef '*name*' in file '*file*' replaces previous definition

If an opdef is defined with the same syntax as a machine instruction, the opdef replaces the machine instruction and CAL issues the same message as shown above.

Opsyn - If an opsyn with the same functional name is defined in more than one binary definition file, the definitions are compared. If the definitions are identical character by character, CAL disregards the duplicate definition and makes one entry for the opsyn from the binary definition files. If the functional name for an opysn is used more than once and the definitions are not identical character by character, CAL uses the definition associated with the last reference to the opsyn name and issues the following diagnostic message:

Opsyn '*name*' in file '*file*' replaces previous definition

If an opsyn is defined with the same name as a pseudo instruction, the opsyn replaces the pseudo instruction and CAL issues the same message as shown above. Pseudos instructions have an internal code that permits CAL to identify them when they are encountered. When an opsyn is used to redefine an existing pseudo instruction, CAL copies the predefined internal code of that pseudo instruction and uses it for identification in the binary definition file.

Micros - If a micro with the same location field name is defined in more than one binary definition file, the micro strings associated with the location field names are compared. If the strings are identical when checked character by character, CAL disregards the duplicate definition and makes one entry for the micro from the binary definition files. If the micro is used more than once and the strings associated with the micro names are not exactly identical, CAL uses the string associated with the last reference to the micro name and issues the following diagnostic message:

Micro '*name*' in file '*file*' replaces previous definition

3. THE CAL PROGRAM

Writing a CAL program requires an understanding of the way a CAL program is organized and how each component functions within the program. This section describes the CAL program and its components.

The following components of a CAL program are discussed in this section.

- Program segment
- Source statement
- Statement editing
- Instructions
- Micros
- Sections

3.1 PROGRAM SEGMENT

A CAL program consists of zero or more segments. A CAL program with zero segments consists of one or more empty files. A file containing one blank line is considered a segment. For example, CAL considers a program with an ident/end sequence that is followed by a blank line to contain two segments. Ordinarily, each segment consists of global definitions, a program module, or a combination of global definitions and a program module. Figure 3-1 illustrates the organization of a CAL program.

3.1.1 PROGRAM MODULE

A program module is the main body of code and resides between the IDENT and END pseudo instructions. (Pseudo instructions are described in more detail later in this section and in section 5.) IDENT marks the beginning of a program module. The END pseudo instruction identifies the end of a module and always terminates a segment. Anything defined between these two pseudo instructions applies only to the program module in which the information resides.

3.1.2 GLOBAL DEFINITIONS

Before the first IDENT pseudo instruction and between program modules (that is, between the END pseudo that terminates one program module and the IDENT that begins the next program module), CAL recognizes sequences

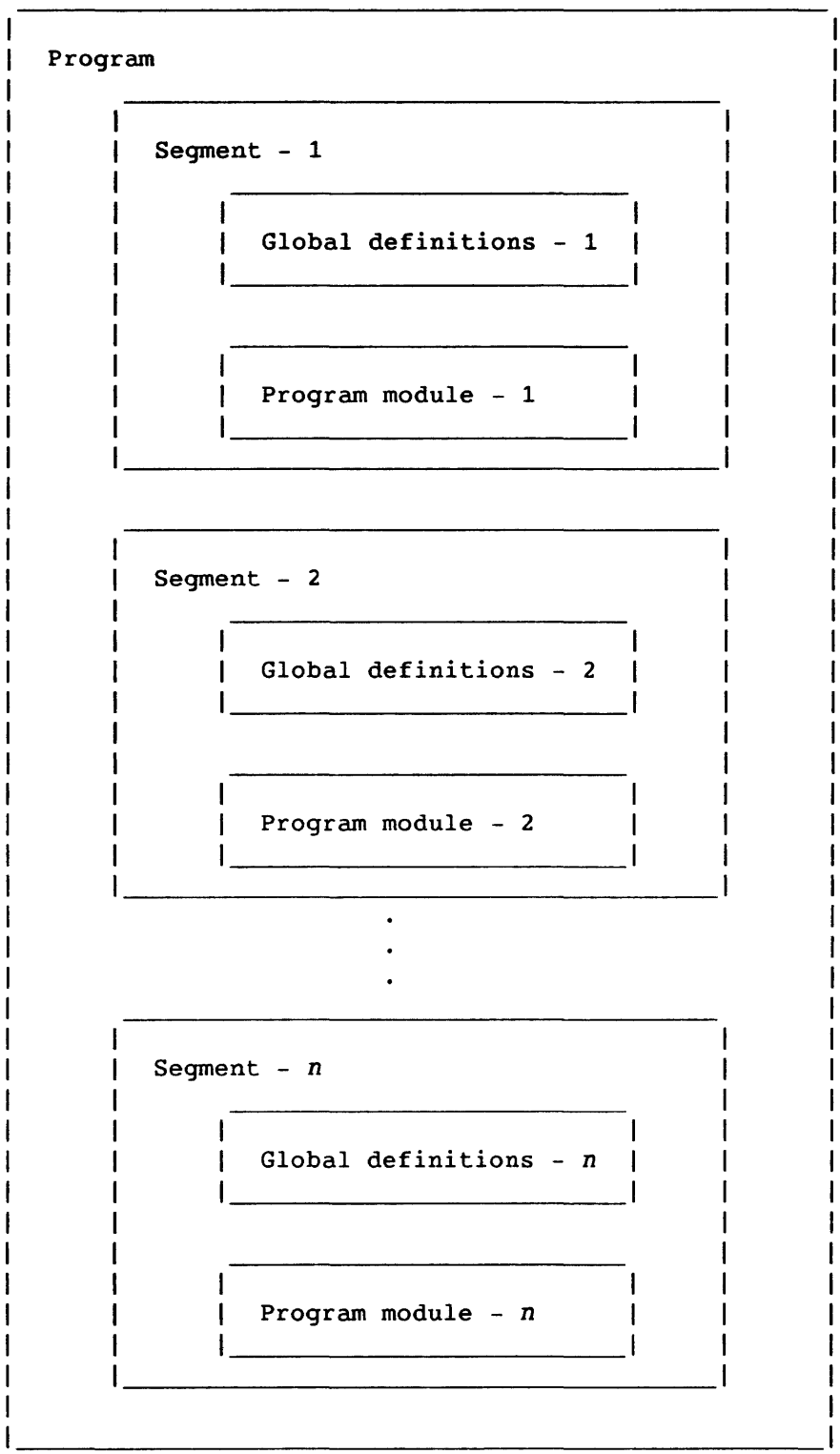

Figure 3-1. Sample Organization of a CAL Program

of instructions that do not generate code but define symbols and assign them values, macros and opdef instructions, and micros. (Opdefs, macros, and micros are described in more detail later in this section and in section 5, Pseudo Instructions.)

Definitions occurring before an IDENT pseudo instruction are considered global and can be referenced without redefinition from within any of the program segments that occur after the definition. Redefinable micros, redefinable symbols, and symbols of the form %%X, where X is zero or more identifier-characters (see appendix A, Instruction Syntax), are exceptions. While they can occur in such sequences, they are local to the segment in which they are defined, are not known to the assembler after the next END pseudo instruction (end of the current segment) is encountere,d and are not included in the cross reference listing. Symbols defined within the global definitions area cannot be qualified.

Example:

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|SYM1    |=         |1         |; Begin segment 1 global
|        |          |          |; SYM1 cannot be redefined
|SYM2    |SET       |2         |; SYM2 equals 2 for this module
|%%SYM3  |=         |3         |; Gone at the end of the module
|%%SYM4  |SET       |4         |; Gone at the end of the module
|        |IDENT     |TEST1     |; Beginning of module 1
|        |S1        |SYM1      |; Register S1 gets 1
|        |S2        |SYM2      |; Register S2 gets 2
|        |S3        |%%SYM3    |; Register S3 gets 3
|        |S4        |%%SYM4    |; Register S4 gets 4
|        |END       |          |; End of segment 1 and module
|        |          |          |; TEST 1
|SYM2    |SET       |3         |; Beginning of segment 2
|%%SYM3  |=         |5         |; Global definitions
|        |IDENT     |TEST2     |; Beginning of module 2
|        |S1        |SYM1      |; Register S1 gets 1
|        |S2        |SYM2      |; Register S2 gets 3
|        |S3        |%%SYM3    |; Register S3 gets 5
|        |S4        |%%SYM4    |; Error; not defined
|        |END       |          |; End of segment 2 and module
|        |          |          |; TEST2
|        |IDENT     |TEST3     |; Beginning of segment 3 and
|        |          |          |; module TEST3
|        |S1        |SYM1      |; Register S1 gets 1
|        |S2        |SYM2      |; Error; not defined
|        |S3        |%%SYM3    |; Error; not defined
|        |END       |          |; End of segment 3 and module
|        |          |          |; TEST3
```

3.2 SOURCE STATEMENT

A CAL program consists of a sequence of source statements. A source statement can be an instruction or a comment. (The assembler lists comments, but they have no effect on the program.)

Although CAL source statements are essentially free field, adoption of formatting conventions provides more uniform and readable listings. CAL supports two formatting conventions, the new format and the old format.

Formal parameters, symbols, names, pseudos, and macro names are case-sensitive. To be recognized, any subsequent references to a previously defined formal parameter, symbol, name, or functional must match the original definition character for character and case for case (uppercase or lowercase). The following are examples of case-sensitivity:

Definition	Reference	Comment
HERE	HERE	Recognized
HERE	Here	Not recognized
PARAM1	param1	Not recognized

When coding in CAL, you can enter statements using both uppercase and lowercase characters according to the following rules.

- Pseudo instructions and mnemonics can be uppercase or lowercase, but not mixed case; case-sensitive.
- Register names can be uppercase, lowercase, or mixed case; case-insensitive.
- Macro names, opdef mnemonics, symbol names, and other names are interpreted as coded; case-sensitive.

CAL supports two source statement formats: new format and old format.

3.2.1 NEW FORMAT

The new format is specified by the FORMAT pseudo or on the invocation statement line of the CAL assembler. For more information about running CAL under the Cray operating systems COS and UNICOS, see section 2, Operating Systems. A source statement using the new format consists of the following four fields.

- Location field
- Result field
- Operand field
- Comment field

If the new format is specified, use the following coding conventions:

Beginning Column	Field
1	Blank or asterisk
1	Location field entry
9	Blank
10	Result field entry
19	Blank
20	Operand field entry
34	Blank
35	Semicolon (indicates comment field)
36	Blank
37	Beginning of comment field

3.2.1.1 Location field

The content of the location field is dependent on the requirements of the result and/or operand fields of each particular source statement. The location field of all machine instructions can optionally contain a symbol. If the location field of a machine instruction contains a symbol, the symbol is set equal to the current address of the location counter.

When the location field is used by an instruction, it begins in column 1 (new format) and is terminated by a blank character. The location field can also contain the * to identify a comment line.

3.2.1.2 Result field

The contents of the result field depends on the particular instruction. The result fields of pseudos and macros must match existing functionals. Machine or opdef instructions can contain one, two, or three subfields.

The subfield can be null, can contain expressions, or can consist of register designators or operators. (Expressions, register designators, and operators are described in section 4, Cray Assembly Language.) The result field begins with the first nonblank character following a nonempty location field and normally ends with one or more blanks or a semicolon. If column 1 is empty, the result field can begin in column 2 or after. A blank result field following a location field produces a listing message.

The detailed syntax for the result field is described using the Backus-Naur Form (BNF) in appendix A, Instruction Syntax.

3.2.1.3 Operand field

The operand field cannot be specified unless it is preceded by a result field. For functionals (pseudos and macro names), the operand field is dependent on the functional specified in the result field.

For symbolic machine instructions, the operand field contains the operation being performed if the instruction is a symbolic instruction. It can, however, contain other information depending on the particular instruction. The syntax of the operand field is identical to that of the result field. Machine or opdef instructions can contain one, two, or three subfields. A subfield can be null, can contain zero or more expressions, or can consist of register designators and operators.

Normally, the operand field begins with the first nonblank character following a nonempty result field and ends with one or more blank characters or a semicolon.

3.2.1.4 Comment field

The comment field contains an explanation of the source statement; it does not generate code. The comment field is optional and can be specified with an asterisk or a semicolon. A semicolon comment can be coded in any blank column including column 1. Generally, a comment that begins in column 1 is specified with an asterisk; otherwise, it is specified by a semicolon. If a semicolon is specified with nothing preceding it, the line is treated as a null instruction followed by a comment. Normally, comment fields are not edited. For more information about editing comment fields, see statement editing in this section.

Example:

```
|Location|Result    |Operand         |Comment
|1       |10        |20              |35
|        |          |                |
|        |ident     |test1           |
|*Asterisk comment  |                |
|        |          |                |; Semicolon comment
```

3.2.2 OLD FORMAT

The old format is specified by the FORMAT pseudo or on the invocation line of the CAL assembler. For more information about running CAL under the Cray operating systems COS and UNICOS, see section 2. A source statement using the old format consists of the following fields:

- Location field
- Result field
- Operand field
- Comment field

If the old format is specified, use the following coding conventions:

Beginning Column	Field
1	Asterisk, or comma
1	Location field entry, left-justified
9	Blank
10	Result field entry, left-justified
19	Blank
20	Operand field entry, left-justified
34	Blank
35	Beginning of comment field

3.2.2.1 Location field

The content of the location field is dependent on the requirements of the result and/or operand fields of each particular source statement. The location field of all machine instructions can optionally contain a symbol. If the location field of a machine instruction contains a symbol, the symbol is set equal to the current address of the location counter.

The location field can also contain the * (column 1 only) to identify a comment line. The location field is not used by all instructions, begins in column 1 or 2 (old format), and is terminated by a blank character.

3.2.2.2 Result field

The result field begins with the first nonblank character following the location field. It cannot begin before column 3 or after column 34. Normally, a blank terminates the result field. The result field has no entry if only blank characters occur between the location field and column 35. A blank result field following a nonblank location field produces a listing message.

The detailed syntax for the result field is described using the Backus-Naur Form (BNF) in appendix A, Instruction Syntax.

3.2.2.3 Operand field

The operand field begins with the first nonblank character following a nonempty result field and ends with one or more blanks. If the result field terminates before column 33, the operand field must begin before column 35; otherwise, the field is considered empty. If the result field extends beyond column 32, however, the operand field must follow at most one blank separator and can begin after column 35.

3.2.2.4 Comment field

The comment field is optional and begins with the first nonblank character following the operand field or if the operand field is empty, does not begin before column 35. If the result field extends beyond column 32 and no operand entry is provided, two or more blanks must precede the comment field. The comment field can be the only field supplied in a statement. If editing is enabled, comments are edited. For more information about editing, see statement editing in this section.

Example:

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|        |IDENT      |TEST1            |
|* An asterisk comment must begin in column 1.
```

3.3 STATEMENT EDITING

CAL processes source statements sequentially from the source file. Statement editing is a form of preprocessing in which CAL deletes or replaces characters before processing the statement as source code. The following types of statement editing are peformed by the assembler:

- Concatenation; the assembler recursively deletes all underscore characters and combines the character that preceded the underscore with the character following the underscore.

- Micro substitution; the assembler replaces a micro name with a predefined character string. The character string replacement is not re-edited.

A macro or opdef definition is not immediately interpreted but is saved and interpreted each time it is called. Before interpreting a statement, CAL performs editing operations. CAL does not perform micro substitution or concatenate lines when editing is disabled. (Editing is disabled by the EDIT pseudo or on the invocation line of the assembler.)

Appending, continuation, and the processing of comments are not affected by the edit invocation statement option.

The following special characters signal micro substitution, concatenation, append, continuation, and comments:

Character	Edit	Description
"*name*"	Yes	Micro; affected by the EDIT pseudo on the invocation statement option (new or old format).
_	Yes	Concatenate; affected by the EDIT pseudo on the invocation statement option (new or old format).
^	No	Append; unaffected by the EDIT pseudo on the invocation statement option (new format).
,	No	Continuation line; unaffected by the EDIT pseudo on the invocation statement option (old format).
*	No	Comment line; unaffected by the EDIT pseudo on the invocation statement option (new or old format).
;	No	Comment line; unaffected by the EDIT pseudo on the invocation statement option (new or old format).

NOTE

When CAL edits "$CMNT", "$MIC", "$CNC", or "$APP" the string name and the double quote marks (" ") are replaced by a previously defined string. For example, when CAL edits "$CMNT", a semicolon is substituted for the micro name $CMNT and the double quote marks (" "). After the substitution occurs, the semicolon is not re-edited and editing continues on the line. Using the predefined "$CMNT" micro permits a comment to be edited. For example:

```
"$CMNT"  Cray Research, Inc.  "$DATE" - "$TIME"
```

is edited as follows:

```
;  Cray Research, Inc.  12/31/85 -  8:15:45
```

The characters to the right of the substituted character are shifted six positions to the left after editing, because the character string substituted for "$CMNT" (;) is six characters shorter than the micro name.

3.3.1 MICRO SUBSTITUTION

You can assign a name to a character string and refer to the character string by its micro name. The CAL assembler searches for quotation marks (") that delimit micro names. The first quotation mark indicates the beginning of a micro name; the second quotation mark identifies the end of a micro name. Before a statement is interpreted, CAL replaces the micro name by the character string comprising the micro. (See micros in subsection 3.5.)

3.3.2 CONCATENATE

The concatenate feature combines characters that are connected by underscore characters. CAL examines each line for the underscore (_) character and deletes it so that the two adjoining columns are linked before the statement is interpreted. The concatenate symbol can be in any column and tells the assembler to concatenate the characters following the last underscore to the character preceding the first underscore.

3.3.3 APPEND

The append feature combines source statements that continue for more than one line and is available only when the new format is specified on the CAL invocation statement.

The append symbol, a circumflex (^), appends one line to another. The append symbol can be in any column. CAL appends the first nonblank character on the next line to the position that contains the circumflex (the circumflex is deleted). A circumflex can be embedded in a micro name. CAL can append a number of lines; the exact number is dependent on memory limitations. Appending is only permitted when the new format is specified.

3.3.4 CONTINUATION

A comma in column 1 indicates a continuation line. Columns 2 through 72 are then a continuation of the previous line. Continuation is only permitted when the old format is specified.

3.3.5 COMMENT

A semicolon (;) in any column (new format) or an asterisk (*) in column 1 indicate a comment line. The assembler lists comment lines, but they have no effect on the program. When a semicolon or an asterisk has an editing symbol after it, the symbol is treated as part of the comment and is not edited. CAL never appends (new format) comment statements with semicolons or asterisks.

NOTE

Asterisk comment statements are not included in macro definitions. To include a comment line in a macro definition, enter an underscore in column 1 of the comment line followed by an asterisk and then the comment. Since editing is disabled at definition time, the statement is inserted. If editing is enabled at expansion time the underscore is edited out and the statement is treated as a comment. For example,

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |MACRO     |                 |
|        |EXAMPLE   |                 |
|* This comment is not included in the definition.
|_* This comment is included in the definition.
|SYM     |=         |1                |
|EXAMPLE |ENDM      |                 |
```

is expanded as follows:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |LIST      |LIS,MAC          |
|        |EXAMPLE   |                 |; Macro call
|* This comment is included in the definition.
|SYM     |=         |1                |
```

3.3.6 ACTUAL STATEMENTS AND EDITED STATEMENTS

CAL statements can be divided into two categories: actual and edited.

An actual statement is the unedited version of a statement that includes any appending of lines. It contains all the editing symbols rather than the results of the editing. If an actual statement has a corresponding edited statement, further processing is done on the edited statement. The following examples show actual and edited statements.

Examples:

1. This is an example of an actual statement.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|LOC     |MCALL     |ARG1,^           |
|        |          |ARG2,^           |
|        |          |ARG3,^           |
|        |          |ARG4,^           |
|        |          |ARG5             |
```

2. An actual statement can have a corresponding edited statement. The edited statement displays the statement without any editing symbols. The following example shows the edited version of the actual statement in example 1.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|LOC     |MCALL     |ARG1,ARG2,ARG3,ARG4,ARG5
```

3. The actual statement in the following example has no corresponding edited statement.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |ENTER     |ARG1,ARG2,ARG3   |; Comments
```

3.4 INSTRUCTIONS

CAL recognizes two types of instructions: assembler-defined and user-defined. Assembler-defined instructions are predefined by CAL. User-defined instructions must be defined by you before you invoke them.

3.4.1 ASSEMBLER-DEFINED INSTRUCTIONS

Two types of assembler-defined instructions are available in CAL: machine instructions and pseudo instructions.

3.4.1.1 Machine instructions

Machine instructions manipulate data by performing such functions as arithmetic operations, memory retrieval and storage, and transfer of control. Each machine instruction can be represented symbolically in CAL. The assembler identifies a machine instruction according to its syntax and generates a binary machine instruction in object code.

The location field of every instruction can contain an optional symbol. If included, an optional symbol has the following qualities: not redefinable, a value equal to the value of the current location counter, an address attribute of parcel, and a relative attribute equal to the relative attribute of the current location counter (absolute, immobile, or relocatable). Refer to section 4, Cray Assembly Language, for more information about symbols and evaluating expressions.

Machine instruction syntax is uniquely defined on the result field alone or on the result and operand fields. The optional location field represents the logical memory location of the instruction. The syntax for machine instructions is described in appendix A, Instruction Syntax.

Each Cray Computer System has its own set of machine instructions. The machine instructions for specific mainframes are discussed in the Symbolic Machine Instruction manuals listed in the preface.

3.4.1.2 Pseudo instructions

Pseudo instructions direct the assembler in its task of interpreting the source statements and generating an object program. CAL has a large complement of pseudo instructions.

Each pseudo instruction has a unique identifier in the result field. The contents of the location and operand fields depend on the pseudo instruction.

Individual pseudo instructions and their formats are described in section 5, Pseudo Instructions. Appendix B, Pseudo Instruction Index, contains an alphabetical list of CAL pseudo instructions.

3.4.2 USER-DEFINED INSTRUCTIONS

The CAL assembler allows you to identify a sequence of instructions to be saved for assembly at a later point in the source program.

CAL recognizes four types of defined sequences: macro, opdef, dup, and echo. Defined sequences come in two classes: permanent and temporary.

A permanent defined sequence (macro or opdef) can be called any number of times after it has been defined. A temporary defined sequence (dup or echo) must be defined before each call. Permanent defined sequences are placed in the source program and assembled when they are called. Temporary defined sequences are assembled immediately after they are defined.

3.5 MICROS

Through the use of micros, you can assign a name to a character string and subsequently refer to the character string by its name. A reference to a micro results in the character string being substituted for the name before assembly of the source statement containing the reference. The CMICRO, MICRO, OCTMIC, and DECMIC pseudo instructions (described in subsection 5.10, Micros) assigns the name to the character string.

A programmer refers to a micro by using the micro name enclosed by quotation marks (") anywhere in a source statement other than within a comment. If column 72 of a line is exceeded as a result of a micro substitution, the assembler creates additional continuation lines. No replacement takes place if the micro name is unknown or if one of the micro quotation marks has been omitted.

When a micro is edited, the source statement in which it is found is changed. Each substitution produces one of the following cases:

- The length of the micro name and the double quote marks is the same as the predefined substitute string. When the micro is edited, the length of the source statement is unchanged.

- The length of the micro name and the double quote marks is greater than the predefined substitute string. When the string is edited, all characters to the right of the edited string shift left the number of spaces equal to the difference between the length of the micro name including the double quote marks and the predefined substitute string.

- The length of the micro name and the double quote marks is less than the predefined substitute string. If column 72 of a line is exceeded as a result of a micro substitution, the assembler creates additional continuation lines. Resulting lines are processed as if they were a single statement.

In the following example, the length of the micro name is equal to the length of the predefined substitute string. A micro named PFX is defined as EQUAL. A reference to PFX is in the location field of the statement as follows:

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|"PFX"TAG|S0        |S1        |; The location of S0 and S1 on
|        |          |          |; the source statement is
|        |          |          |; unchanged.
```

When the line is interpreted, CAL substitutes the definition (EQUAL) for "PFX" producing the following line.

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|EQUALTAG|S0        |S1        |; The location of S0 and S1 on
|        |          |          |; the source statement is
|        |          |          |; unchanged.
```

In the following example, the length of the micro name is greater than the length of the predefined substitute string. A micro named PFX is defined as LESS. A reference to PFX is in the location field of the statement as follows:

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|"PFX"TAG|S0        |S1        |; Since LESS is one character
|        |          |          |; shorter than the micro string
|        |          |          |; name "PFX", the values in the
|        |          |          |; result and operand fields are
|        |          |          |; shifted one space to the left.
```

Before the line is interpreted, CAL substitutes the definition (LESS) for "PFX" producing the following line.

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|LESSTAG S0        |  S1       |; Since LESS is one character
|        |          |           |; shorter than the micro string
|        |          |           |; name "PFX", the values in the
|        |          |           |; result and operand fields are
|        |          |           |; shifted one space to the left.
```

In the following example, the length of the micro name is less than the length of the predefined substitute string. A micro named pfx is defined as greater. A reference to pfx is in the location field of the following statement:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|"pfx"tag|s0        |s1         |; Since greater is two
|        |          |           |; characters longer than the
|        |          |           |; micro string name "pfx", the
|        |          |           |; values in the result and
|        |          |           |; operand fields are shifted
|        |          |           |; two spaces to the right.
```

Before the line is interpreted, CAL substitutes the predefined string greater for "pfx". Since the predefined substitute string is two characters longer than micro name, the fields to the right of the substitution are shifted two characters to the right producing the following:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|greatertag s0      |  s1       |; Since greater is two
|        |          |           |; characters longer than the
|        |          |           |; micro string name "pfx", the
|        |          |           |; values in the result and
|        |          |           |; operand fields are shifted
|        |          |           |; two spaces to the right.
```

One or more micro substitutions can occur between the beginning and ending quotation marks of a micro. These substitutions create a micro name that is substituted, along with the surrounding quotation marks, for the corresponding micro string. Substitutions of this type are referred to as embedded micros. An embedded micro consists of a {, a micro name, and a } and is specified as follows:

{*micro-name*}

When a micro containing one or more embedded micros is encountered, CAL edits all of the embedded micros within the micro until a micro name is recognized or until the micro name is determined to be illegal (undefined or exceeding the maximum allowable string length of eight characters). When an illegal micro is encountered, CAL issues an appropriate message and terminates the editing of the micro. An embedded micro can itself contain one or more embedded micros.

The following example includes valid and invalid defined embedded micros:

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|index   |micro      |\1\              |; Assigns literal value to index
|null    |micro      |\\               |; Assigns literal value to null
|array"index" micro \Some string\     |
|array1 micro \Some string\†          |
|        |           |                 |
|_* "array1" - This is an explicit reference.
|* Some string - This is an explicit reference.†
|        |           |                 |
|_* "array""index" - This is invalid, because 'array' was not defined.
|* "array"1 - This is invalid, because 'array' was not defined.†
|        |           |                 |
|_* "array{index}" - This is an example of an embedded micro.
|* Some string - This is an example of an embedded micro.†
|        |           |                 |
|_* "{null}array{index}" - This is an example of two embedded micros.
|* Some string -  This is an example of two embedded micros.†
```

CAL places no restrictions on the number of recursions that are necessary to identify a micro name. The following example demonstrates the unlimited recursive editing capablility of CAL on embedded micros:

† Edited by CAL

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|index   |micro      |\1\              |; Assigns literal value to index
|null    |micro      |\\               |; Assigns literal value to null
|array"index" micro \Some string\     |
|array1 micro \Some string\†          |
|        |           |                 |
|_* "{nu{n{null}u{null}ll}ll}ar{null{null}}ray{ind{null}ex}" - Micro
|* Some string - Micro†               |
```

CAL issues an informative message with a priority of warning or error when an invalid micro name is specified. If a micro name is recognized as being invalid before any editing has begun, a message with a priority of warning is issued. If any embedded micro has been edited and the resulting string is an invalid micro name, a message with a priority of error is issued.

The following example demonstrates how CAL assigns priorities to messages when an invalid micro is encountered:

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|identity micro      |\The substitute string for this example\
|null    |micro      |\\               |; Assigns literal value to null
|        |           |                 |
|_* "identity{null}" - This is a valid micro.
|* The substitute string for this example - This is a valid micro.†
|        |           |                 |
|* The following micro is invalid, because the maximum micro name
|* length of eight characters is exceeded.  When a micro name is
|* identified as being invalid before editing occurs, a message with a
|* priority of warning is issued.     |
|*       |           |                 |
|_* "identity9{null}" - This is an invalid micro.
|* "identity9 - This is an invalid micro.†
|        |           |                 |
|* The following micro is invalid, because the maximum micro name
|* length of eight characters is exceeded.  When a micro name is
|* identified as being invalid after editing occurs, a message with a
|* priority of error is issued.       |
|        |           |                 |
|_* "id{null}entity9{null}" - This is an invalid micro.
|* "identity9" - This is an invalid micro.†
```

† Edited by CAL

3.6 SECTIONS

A CAL module can be divided into blocks of memory called sections. By dividing a program into sections, you can conveniently separate executable sequences of code from nonexecutable data. As assembly of a program proceeds, you can explicitly or implicitly assign code to specific sections or reserve areas of a section. The assembler assigns locations in a section consecutively as it encounters instructions or data destined for that particular memory section.

The main and literals sections are used for implicitly assigned code. CAL maintains a stack of section names assigned by the SECTION† pseudo instruction. All sections are passed directly to the loader with the exception of stack sections.

Sections can be local or common. A local section is available to the CAL program module in which it resides; a common section is available to another CAL program module.

To explicitly assign code to a section, use the SECTION† pseudo instruction. The SECTION pseudo instruction can be specified for CRAY-2, CRAY X-MP, or CRAY-1 Computer Systems.

3.6.1 LOCAL SECTIONS

A local section is a block of code that is useable only by the program module in which it resides. CAL uses three types of local sections.

- Main section
- Literals section
- Sections defined by the SECTION pseudo

When a SECTION pseudo instruction is used, every SECTION type except COMMON, DYNAMIC, and TASKCOM are local. For more detailed information about SECTION types, see the SECTION pseudo in subsection 5.4, Section Control.

3.6.1.1 Main section

The main section is initiated by the IDENT pseudo and is always the first section in a program module. This section is used for all local code other than that generated by the occurrence of a literal reference or code between two SECTION† pseudo instructions.

† The BLOCK and COMMON pseudo instructions can also be used to implicitly or explicitly assign code to memory blocks for CRAY X-MP and CRAY-1 Computer Systems. BLOCK and COMMON are not supported for the CRAY-2 Computer System.

Generally, sections may or may not have names but must be assigned types and locations. The main section's default name is always empty. The defaults for type and location are MIXED and CM, respectively. For more information about section name, MIXED and CM, see the SECTION pseudo in subsection 5.4, Section Control.

3.6.1.2 Literals section

The first use of a literal value in an expression causes the assembler to store the data item in a literals section. For more information about literals, see section 4, Cray Assembly Language. Data is generated in the literals section implicitly by the occurrence of a literal. Explicit data generation or memory reservation is not allowed in the literals section.

3.6.1.3 Sections defined by the SECTION pseudo

When a SECTION† pseudo instruction is used, all code generated or memory reserved (other than literals) from the occurrence of one SECTION pseudo instruction up to the occurrence of the next SECTION pseudo instruction is assigned to the designated section. Until the first SECTION pseudo instruction is specified, the main section is used. An exception to these conditions can occur if the ORG pseudo instruction is specified. Specifying the ORG pseudo instruction may cause the placement of code or memory reservations to be different from the currently specified working section.

Although the BLOCK and COMMON pseudos can be specified with CRAY X-MP or CRAY-1 Computer System, the SECTION pseudo is recommended for use with all types of Cray Computer Systems (CRAY-2, CRAY X-MP, and CRAY-1), because it has all of the capabilities of the BLOCK and COMMON pseudos in addition to many other capabilities.

When a section is released (see SECTION * in section 5, Pseudo Instructions), the type and location of the previous section is used. When the number of sections released is equal to or greater than the number specified, CAL uses the defaults of the main section for type (MIXED) and location (CM).

A section with the same name, type, and location used in different areas of a program is recognized as the same section.

---

† The SECTION pseudo replaces the BLOCK pseudo instruction. SECTION can be used in any of the ways that BLOCK was previously used. BLOCK is not supported on the CRAY-2 Computer System.

3.6.2 COMMON SECTIONS

When a SECTION† pseudo instruction is used with a type of COMMON, DYNAMIC, or TASKCOM, all code generated (other than literals) or memory reserved from the occurrence of one SECTION instruction up to the occurrence of the next SECTION instruction is assigned to the designated common, dynamic, or task common section. At program end, each common section is identified to the loader by its SECTION name and is available for reference by another program module. An exception to these conditions can occur if the ORG pseudo instruction is specified. Specifying the ORG pseudo instruction may cause the placement of code or memory reservations to be different from the currently specified working section.

If a common section is named, the identifier in the location field that names the section must be unique within the module in which it is defined. Even if a section is assigned a type (COMMON, DYNAMIC, or TASKCOM) that is different from the type of a previously defined section, it cannot be assigned the name of a previously defined section within the same module. If duplicate location field names are specified, a message with a priority of error is issued.

3.6.3 SECTION STACK BUFFER

CAL maintains a stack buffer that contains a list of the sections that have been specified. Each time a SECTION†† pseudo instruction names a new section, CAL adds the name of the section to the list and identifies the new section as the current section. CAL remembers the order that sections are specified. An entry is deleted from the list each time a SECTION pseudo contains an *. When an entry is deleted, the name, location, and type of the section specified before the deleted section is enabled.

The first section on the list is the last section to be deleted from the list. If the program contains more SECTION * instructions††† than there are entries, the assembler uses the main section.

For each section used in a program, CAL maintains an origin counter, a location counter, and a bit position counter. When a section is first established or its use is resumed, CAL uses the counters for that section.

---

† The SECTION pseudo replaces the COMMON pseudo instruction. SECTION can be used in any of the ways that COMMON was previously used. COMMON is not supported on the CRAY-2 Computer System.

†† The BLOCK and COMMON pseudo instructions can also be used to name sections. BLOCK and COMMON are not supported on the CRAY-2 Computer System.

††† The BLOCK * and COMMON * instructions replaces the current section with the most recent previous section that was specified by the BLOCK and COMMON pseudo instructions.

The following example illustrates specifying sections, the current section in effect, and deleting sections. The example includes the QUAL pseudo. For a detailed description of the QUAL pseudo see subsection 5.3, Mode Control.

Example:

```
|Location|Result    |Operand        |Comment
|1       |10        |20             |35
|        |          |               |
|        |IDENT     |STACK          |; The IDENT statement puts the
|        |          |               |; first entry on the list of
|        |          |               |; qualifiers; this entry starts
|        |          |               |; the symbol table for
|        |          |               |; unqualified symbols.
|SYM1    |=         |1              |; SYM1 is relative to the main
|        |          |               |; section.
|        |QUAL      |QNAME1         |; Second entry on the list of
|        |          |               |; qualifiers.
|SYM2    |=         |2              |; SYM is the first entry in the
|        |          |               |; symbol table for QNAME1.
|SNAME   |SECTION   |MIXED          |; SNAME is the second entry on
|        |          |               |; the list of sections.
|        |MLEVEL    |ERROR          |; Reset message level to error
|        |          |               |; eliminate warning level
|        |          |               |; messages.
|SYM3    |=         |*              |; SYM3 is the second entry in
|        |          |               |; the
|        |          |               |; symbol table for QNAME1 and is
|        |          |               |; relative to the SNAME section.
|        |MLEVEL    |*              |; Reset message level to default
|        |          |               |; in effect before the MLEVEL
|        |          |               |; specification.
|        |SECTION   |*              |; SNAME is deleted from the list
|        |          |               |; of sections.
|SYM4    |=         |4              |; SYM4 is the third entry in the
|        |          |               |; symbol table for QNAME1 and is
|        |          |               |; relative to the main section.
|        |QUAL      |QNAME2         |; Third entry on the list of
|        |          |               |; qualifiers.
|SYM5    |=         |5              |; SYM5 is the first entry in the
|        |          |               |; symbol table for QNAME2.
|SYM6    |=         |/QNAME1/SYM2   |; SYM6 gets SYM2 from the symbol
|        |          |               |; table for QNAME1 even though
|        |          |               |; QNAME1 is not the current
|        |          |               |; qualifier in effect.
|        |QUAL      |*              |; QNAME2 is removed as the
|        |          |               |; current qualifier name SYM7
|SYM7    |=         |6              |; is the fourth entry in the
|        |          |               |; symbol table for QNAME1.
```

Example: (continued)

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |QUAL      |*           |; QNAME1 is removed as the
|        |          |            |; current qualifier name
|SYM8    |=         |7           |; Second entry in the symbol
|        |          |            |; table for unqualified symbols.
```

3.6.3.1 Origin counter

The origin counter controls the relative location of the next word to be assembled or reserved in the section. It is possible to reserve blank memory areas simply by using either the ORG or BSS pseudo instructions to advance the origin counter. When the special element *O is used in an expression, the assembler replaces it with the current parcel-address value of the origin counter for the section in use. Special elements are described in section 4, Cray Assembly Language. W.*O can be used to obtain the word-address value of the origin counter. For more information about the W. prefix, see section 4.

3.6.3.2 Location counter

The location counter is normally the same value as the origin counter and is used by the assembler for defining symbolic addresses within a section. The counter is incremented whenever the origin counter is incremented. It is possible to use the LOC pseudo instruction to adjust the location counter so that it differs in value from the origin counter or so that it refers to the address relative to a section other than the one currently in use. When the special element * is used in an expression, the assembler replaces it by the current parcel-address value of the location counter for the section in use. W.* can be used to obtain the word-address value of the location counter.

3.6.3.3 Word-bit-position counter

As instructions and data are assembled and placed into a word, CAL maintains a pointer indicating the next available bit within the word currently being assembled. This pointer is known as the word-bit-position counter. It is 0 when a new word is begun and is incremented by 1 for each completed bit in the word. Its maximum value is 63 for the right-most bit in the word. When a word is completed, the origin and location counters are incremented by 1 and the word-bit-position counter is reset to 0 for the next word.

When the special element *W is used in an expression, the assembler replaces it with the current value of the word-bit-position counter. The normal advancement of the word-bit-position counter is in increments of 16, 32, and 64 as 1-parcel and 2-parcel instructions or words are generated. This normal advancement can be altered, however, through use of the BITW, BITP, DATA, and VWD pseudo instructions.

3.6.3.4 Force word boundary

The assembler completes a partial word and sets the word-bit-position and parcel-bit-position counters to 0 if either of the following conditions is true:

- The current instruction is an ORG, LOC, BSS, BSSZ, CON, or ALIGN pseudo instruction.
- The current instruction is a DATA or VWD pseudo instruction and the instruction has an entry in the location field.

3.6.3.5 Parcel-bit-position counter

In addition to the word-bit-position counter, CAL also maintains a counter that points to the next bit to be assembled in the current parcel. This pointer is known as the parcel-bit-position counter. It is 0 when a new parcel is begun and advances by 1 for each completed bit in the parcel. Its maximum value is 15 for the right-most bit in a parcel. When a parcel is completed, the parcel-bit-position counter is reset to 0.

When the special element *P is used in an expression, CAL replaces it with the current value of the parcel-bit-position counter.

The parcel-bit-position counter will be set to 0 following assembly of most instructions. The pseudo instructions BITW, BITP, DATA, and VWD can cause the counter to be nonzero.

3.6.3.6 Force parcel boundary

The assembler completes a partially filled parcel and sets the parcel-bit-position counter to 0 if the current instruction is a symbolic machine instruction.

4. CRAY ASSEMBLY LANGUAGE

This section presents the general rules and statement syntax for coding a Cray Assembly Language (CAL) program. CAL syntax is described using Backus-Naur Form (BNF). For a complete listing of CAL BNF and a description of BNF notation, see appendix A, Instruction Syntax, and section 1, Introduction, respectively. This section describes the following instruction syntax:

- Register designators
- Names
- Symbols
- Data
- Special elements
- Element prefixes for symbols, constants, or special elements
- Expressions
- Expression evaluation
- Expression attributes

4.1 REGISTER DESIGNATORS

Register designators are used in symbolic machine instructions and opdefs to identify which register is used for an operation. Each Cray Computer System supports all or a subset of the following types of operating registers. The register is defined as follows:

```
register ::= complex-register | simple-register .
```

4.1.1 COMPLEX REGISTERS

A complex register is a member of a set of registers that are identical in function and architecture. These registers are identified by register names that are comprised of a letter followed by an octal number or a constant. For example, register S1 can be specified from the group of registers known as the S registers.

The A, B, SB, SM, SR, ST, S, T, and V registers can be designated as complex registers. Any complex register that is available on a Cray Computer System can be specified by CAL. The exact combination of registers available in symbolic machine instructions to CAL depends on the Cray Computer System for which CAL is targeting code.

CAL accepts register mnemonics that are specified in uppercase, lowercase, or mixed case. Complex registers can be specified with up to four octal digits. Although CAL does not restrict the way designators are entered, the requirements of the Cray Computer System for which CAL is targeted can be more specific.

Some register designators have letter prefixes that have special meaning to the assembler. The prefixes and their meanings are listed in the appropriate Cray symbolic machine instruction manual. For more information about the complex registers available with your Cray Computer System, see the CRAY-2 Computer System Functional Description, publication HR-2000, and the Symbolic Machine Instructions Reference Manual, publication SR-0085.

Complex registers are defined as follows:

```
complex-register ::= complex-register-mnemonic register-designator .

complex-register-mnemonics ::= "A" | "B" | "SB" | "SM" | "SR" | "ST" |
                               "S" | "T" | "V" .†

register-designator ::= octal-digit [ octal-digit [ octal-digit
                       [ octal-digit ] ] ] |
                       "." integer-constant | "." symbol .
```

Examples:

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|SyM     |=         |*         |; CAL permits mixed case
|        |A1        |SyM       |; in any combination with the
|        |          |          |; following restriction:
|        |          |          |; matching names must be entered
|        |          |          |; in the same manner.
|REG     |=         |3         |
|        |A.REG     |A1        |; Register A3 gets the contents
|        |          |          |; of A1
|        |S1        |S2        |; Register S1 gets the contents
|        |          |          |; of S2
```

† Uppercase, lowercase, and mixed case in any combination is permitted by CAL.

4.1.2 SIMPLE REGISTERS

A simple register has a predefined function that cannot be redefined. These registers are identified by register names that are comprised of letters only.

The CA, CE, CI, CL, MC, RT, SB, SM, VL, VM, and XA registers have been designated as simple registers.

For more information about the simple registers available with your Cray Computer System, see the CRAY-2 Computer System Functional Description, publication HR-2000, and the Symbolic Machine Instructions Reference Manual, publication SR-0085.

The simple-register designator is defined as follows:

```
simple-register ::= simple-register-mnemonic .

simple-register-mnemonic ::= "CA" | "CE" | "CI" | "CL" | "MC" | "RT" |
                             "SB" | "SM" | "VL" | "VM" | "XA" .†
```

Example:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |S1        |RT               |; Register S1 gets the contents
|        |          |                 |; of the RT register
```

4.2 NAMES

A name is a one- to eight-character identifier. The first character (initial-identifier-character) must be alphabetic (A through Z), a dollar sign ($), a percent sign (%), or an at sign (@). Characters 2 through 8 (identifier-characters) can also be decimal digits (0 through 9).

Unlike a symbol, a name does not have a value or an attribute associated with it and cannot be used in expressions.

† CAL permits uppercase, lowercase, and mixed case in any combination.

Names are used to identify the following types of information.

- Program modules
- Sections
- Macro instructions
- Micro character strings
- Conditional sequences
- Duplicated sequences

Names are defined as follows:

```
name ::= identifier .
```

Different types of names do not conflict with each other or with symbols. For example, a micro can have the same name as a macro and a program module can have the same name as a section.

Examples of valid and invalid names:

Valid	Comment
count	Lowercase is permitted
@ADD	@ legal beginning character
ABC5	Combinations of letters and digits are legal if the first character is an initial-identifier-character

Invalid	Comment
9knt	Begins with a number
JOHNJONES	More than 8 characters
+YZ3	Begins with +

4.3 SYMBOLS

A symbol is a 1- to 8-character identifier that has a value and attributes associated with it and can be used in expressions. A symbol can be used in the following ways:

- When a symbol is in the location field of a source statement, the symbol is being defined for use in the program. When a symbol is defined, it assumes a value and certain characteristics called attributes.

- When a symbol is in the operand or result field of a source statement, the symbol is being referenced.

- Loader linkage

A symbol can be local or global depending on where the symbol is defined. That is, a symbol can be used within a single program module (local) or by a number of program segments (global). (See global definitions, section 3, The CAL Program.) A symbol can also be made unique to a code sequence (see qualified symbols, this section).

Symbols of the following form %% are generated by CAL:

    %%*nnnnnn*

where *n* is a decimal digit.

%% symbols are discarded at the end of a program segment regardless of whether or not the symbol is redefinable or defined in the global definitions part.

For more detailed information about symbols generated by CAL, see the description of the LOCAL pseudo in subsection 5.12, Defined Sequences.

CAL issues a warning message if a symbol is a valid identifier and is defined as one of the following registers reserved by CAL:

```
register ::= complex-register | simple-register .

complex-register ::= complex-register-mnemonic register-designator .

complex-register-mnemonics ::= "A" | "B" | "SB" | "SM" | "SR" | "ST" |
                               "S" | "T" | "V" . †

register-designator ::= octal-digit [ octal-digit [ octal-digit
                        [ octal-digit ] ] ] .

simple-register ::= simple-register-mnemonic .

simple-register-mnemonic ::= "CA" | "CE" | "CI" | "CL" | "MC" | "RT" |
                             "SB" | "SM" | "VL" | "VM" | "XA" .†
```

A symbol can be used in the following ways:

- Specified as unqualified or qualified
- Defined, that is, associated with a value and attributes

† Uppercase, lowercase, and mixed case in any combination is permitted by CAL.

- Assigned the following attributes: address, relative, and redefinable

- Referenced by using the value instead of the symbol itself

4.3.1 SYMBOL SPECIFICATION

Symbols can be specified as unqualified or qualified and are defined as follows:

```
symbol ::= unqualified-symbol | qualified-symbol .
```

4.3.1.1 Unqualified symbol

An unqualified symbol is a 1- to 8-character identifier that identifies a value and its associated attributes (see following description of symbol attributes). The initial-identifier-character of a symbol must be a letter (upper or lowercase A-Z), a dollar sign ($), a percent sign (%), or an at sign (@). The characters that follow the initial-identifier-character (identifier-characters) can also be decimal digits (0 through 9).

A warning message is issued if a symbol is defined with an identifier that matches the syntax of a register.

Unqualified symbols can be referenced as follows:

- Unqualified symbols, defined in an unqualified code sequence and referenced from within the unqualified code sequence, can be referenced without qualification.

- Unqualified symbols can be referenced within the current qualifier without qualification if the symbol has not been defined within the current qualifier.

- Unqualified symbols can be referenced from within the current qualifier using the form *//symbol*.

Unqualified symbols are defined as follows:

```
unqualified-symbol ::= identifier .
```

Example:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |IDENT     |TEST        |
|SYM1    |=         |*           |; SYM1 has a value equal to the
|        |          |            |; location counter
|        |A1        |SYM1        |; Register A1
|SYM2    |SET       |2           |; SYM2 is redefinable
|SYM3    |=         |3           |; SYM3 is not redefinable
|        |END       |            |
```

4.3.1.2 Qualified symbols

A symbol that is not a global symbol can be made unique to a code sequence by specifying a symbol qualifier that is to be appended to all symbols defined within the sequence. The QUAL pseudo instruction qualifies symbols (see QUAL pseudo instruction in subsection 5.3, Mode Control). Qualified symbols must be defined with respect to following rules:

- A qualified symbol cannot be defined with a label that is reserved for complex or simple registers. A warning message is issued if a symbol is defined with such a label.
- Symbols can be qualified in a program module only.
- Symbols can never be qualified in the global definitions part of a program.

Qualified symbols can be referenced as follows:

- If a qualified symbol (defined in a code sequence) is referenced from within a sequence, it can be referenced without qualification.
- If a qualified symbol is referenced outside of the code sequence in which it was defined, it must be referenced in the form */qualifier/symbol*, where *qualifier* and *symbol* are one- to eight-character identifiers and are defined by a QUAL pseudo instruction.

Qualified symbols are defined as follows:

```
qualified-symbol ::= "/" [ identifier ] "/" identifier .
```

Example:

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|        |IDENT     |TEST         |;
|SYM1    |=         |1            |; Assignment
|        |QUAL      |NAME1        |; Declare qualifier name
|SYM1    |=         |2            |; Qualified symbol SYM1
|        |S1        |SYM1         |; Register S1 gets 2
|        |          |             |; (qualified SYM1)
|        |S1        |//SYM1       |; Register S1 gets 1
|        |          |             |; (unqualified SYM1)
|        |S1        |/NAME1/SYM1  |; Register S1 gets 2
|        |          |             |; (qualified SYM1)
|        |QUAL      |*            |; Pop the top of the qualifier
|        |          |             |; stack
|        |S1        |SYM1         |; Register S1 gets 1
|        |S1        |//SYM1       |; Register S1 gets 1
|        |S1        |/NAME1/SYM1  |; Register S1 gets 2
|        |END       |             |
```

4.3.2 SYMBOL DEFINITION

A symbol is defined by assigning it a value and attributes. A symbol's value and attributes depend on how the symbol is used in the program. The assignment can occur in the following three ways.

- When a symbol is used in the location field of a symbolic machine instruction or certain pseudo instructions, it is defined as follows:

  - Having the address of the current value of the location counter (counters are described in section 3, The CAL Program)

  - Having parcel-address or word-address attributes

  - Being absolute, immobile, or relocatable

  - Not redefinable

- A symbol used in the location field of a symbol-defining pseudo instruction is defined as having the value and attributes derived from an expression in the operand field of the instruction. Some symbol-defining pseudo instructions cause the symbol to have a redefinable attribute. When a symbol is redefinable, a redefinable pseudo instruction must be used to define the symbol the second time. Redefinition of the symbol causes it to be assigned a new value and attributes.

- A symbol can be defined as external to the current program module. A symbol is external if it is defined in a program module other than the module currently being assembled. The true value of an external symbol is not known within the current program module.

Examples:

Each of the following is an example of a symbol.

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|START   |=         |*           |; The symbol START has the
|        |          |            |; current value of the
|        |          |            |; location counter and cannot be
|        |          |            |; redefined.
|PARAM   |SET       |D'18        |; The symbol PARAM is equal to
|        |          |            |; the decimal value 18 and can
|        |          |            |; be redefined.
|        |EXT       |SECOND      |; Identifies SECOND as an
|        |          |            |; external symbol
```

4.3.3 SYMBOL ATTRIBUTES

When a symbol is defined, it assumes two or more attributes. These attributes fall into three categories:

- Address
- Relative
- Redefinable

Every symbol is assigned one attribute from each of the first two categories. Whether or not a symbol is assigned the redefinable attribute depends on how the symbol is used. Every symbol has a value of up to 64 bits associated with it.

4.3.3.1 Address attributes

Each symbol is assigned one of the following address attributes:

- Word address
- Parcel address
- Value

Word address - A symbol is assigned a word-address attribute if it appears in the location field of a pseudo instruction, such as a BSS or BSSZ, that defines words; if is equated to an expression having a word-address attribute; or if the word is explicitly stated in the operand field of an EXT pseudo.

Parcel address - A symbol is assigned a parcel-address attribute if it appears in the location field of a symbolic machine instruction or certain pseudo instructions; is equated to an expression having a parcel-address attribute; or if parcel is explicitly stated in the operand field of an EXT pseudo.

Value - A symbol has a value attribute if it does not have word-address or parcel-address attributes, or if value is explicitly stated in the operand field of an EXT pseudo. All globally defined symbols have an address attribute of value.

4.3.3.2 Relative attributes

Every symbol is assigned one of the following relative attributes:

- Absolute
- Immobile
- Relocatable
- External

Absolute - A symbol is assigned the relative attribute of absolute when the current location counter is absolute and it appears in the location field of a machine instruction, BSS pseudo instruction, or data generation pseudo instruction such as BSSZ or CON. A symbol is also absolute if it is equated to an expression that is absolute. All globally defined symbols have a relative attribute of absolute. The symbol is only known at assembly time.

Immobile - A symbol is assigned the relative attribute of immobile when the current location counter is immobile and it appears in the location field of a machine instruction, BSS pseudo instruction or data generation pseudo instruction such as BSSZ or CON. A symbol is also immobile if it is equated to an expression that is immobile. The symbol is only known at assembly time.

Relocatable - A symbol is assigned the relative attribute of relocatable when the current location counter is relocatable and it appears in the location field of a machine instruction, BSS pseudo instruction, or data generation pseudo instruction such as BSSZ or CON. A symbol is also relocatable if it is equated to an expression that is relocatable.

External - A symbol is assigned the relative attribute of external when it is defined by an EXT pseudo instruction. An external symbol defined in this manner is entered in the symbol table with a value of 0. You can specify the address attribute of an external symbol as value (V), parcel (P), or word (W); the default is value.

A symbol is also assigned the relative attribute of external if it is equated to an expression that is external. Such a symbol assumes the value of the expression and can have an attribute of parcel address, word address, or value.

The assignment of an unknown variable with a register at assembly time, can be made by use of a symbol with a relative attribute of external. In the following example, register s1 is loaded with variable ext1 at assembly time.

Example:

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|        |ident     |test1     |
|        |ext       |ext1      |; Variable ext1 is defined as
|        |          |          |; an external variable
|        |s1        |ext1      |; ext1 transmits value to
|        |          |          |; register s1
|        |end       |          |
|        |ident     |test2     |
|        |entry     |ext1      |
|ext1    |=         |3         |; When the two modules are
|        |          |          |; linked, register S1 gets 3.
|        |end       |          |
```

4.3.3.3 Redefinable attributes

In addition to its other attributes, a symbol is assigned the attribute of redefinable if it is defined by the SET or MICSIZE pseudo instructions. A redefinable symbol can be defined more than once in a program segment and can have different values and attributes at different times during an assembly. When such a symbol is referenced, its most recent definition is used by the assembler. All redefinable symbols are discarded at the end of a program segment without regard to whether they were defined in the global definitions or not.

Examples:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |IDENT     |TEST        |
|SYM1    |=         |1           |; Not redefinable
|SYM2    |SET       |2           |; Redefinable
|SYM1    |SET       |2           |; Error; SYM1 previously
|        |          |            |; defined as 1
|SYM2    |SET       |3           |; Redefinable
|        |END       |            |
```

4.3.4 SYMBOL REFERENCE

When a symbol is in a field other than the location field, the symbol is being referenced. Reference to a symbol within an expression causes the value and attributes of the symbol to be used in place of the symbol. Symbols may also be found in the operand fields of pseudos.

A symbol reference within an expression can contain a prefix which causes the usual value and attributes associated with the symbol to be altered according to the prefix. The prefix affects only the specific reference with which it occurs. For details, refer to subsection 4.6, Element Prefixes for Symbols, Constants, or Special Elements.

Examples:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |S1        |SYM1+1      |; Register S1 gets the location
|        |          |            |; of SYM1+1.  SYM1+1 is an
|        |          |            |; example of a symbol in an
|        |          |            |; operand
|        |          |            |; field used as an expression.
|        |IFA       |DEF,SYM1    |; Symbols can also be used
|        |          |            |; outside
|        |          |            |; of an expression.  In this
|        |          |            |; instance, SYM1 is not used
|        |          |            |; within
|        |          |            |; an expression; it is a symbol.
```

4.4 DATA

Some instructions manipulate data. CAL instructions can use data that is specified in any of the following forms:

- Constants
- Data items
- Literals

4.4.1 CONSTANTS

Constants are defined as follows:

```
constant ::= floating-constant | integer-constant | character-constant .
```

4.4.1.1 Floating-constant

A floating-constant is evaluated as a 1- or 2-word quantity, depending on the precision specified. (See the floating-point data formats figures in the approriate Symbolic Machine Instruction Manual.)

The floating-constant is defined as follows:

```
floating-constant ::= [ decimal-prefix ] floating-decimal
                      [ binary-scale decimal-integer ] ] .
```

decimal-prefix
Numeric base used for the floating-decimal and/or the decimal-integer. D' or d' specifies a decimal-prefix and is the only prefix available for a floating-constant.

floating-decimal
A floating-decimal can be one of the following:

A *decimal-integer* followed by a *decimal-fraction* with an optional *decimal-exponent* and *decimal-integer;* for example:

n.n or *n.n*E*n* or *n.n*E±*n* or *n.n*D*n* or *n.n*D±*n*

A *decimal-integer* followed by a "." with a *decimal-exponent* and *decimal-integer;* for example:

n. or *n.*E*n* or *n.*E±*n* or *n.n*D*n* or *n.n*D±*n*

A *decimal-integer* followed by a *decimal-exponent* and *decimal-integer*; for example:

*n*E*n* or *n*E±*n* or *n*D*n* pr *n*D±*n*

A *decimal-fraction* followed by an optional *decimal-exponent* and *decimal-integer*; for example:

.*n* or .*n*E*n* or .*n*E±*n* or .*n*D*n* or .*n*D±*n*

A *decimal-integer* is a nonempty string of decimal digits. A *decimal-integer decimal-fraction* is a nonempty string of decimal digits representing a whole number, a mixed number, or a fraction.

decimal-exponent

The power of 10 by which the integer and/or *fraction* is to be multiplied; indicates whether the constant is to be single precision (E or e; one 64-bit word) or double precision (D or d; two 64-bit words). *n* is an integer in the base specified by *prefix*.

If no *decimal-exponent* is provided, the constant occupies one word. *decimal-exponents* are defined as follows:

E*n*	Positive decimal exponent, single precision
E+*n*	Positive decimal exponent, single precision
E-*n*	Negative decimal exponent, single precision
D*n*	Positive decimal exponent, double precision
D+*n*	Positive decimal exponent, double precision
D-*n*	Negative decimal exponent, double precision

binary-scale decimal-integer

The *integer* and/or *fraction* is to be multiplied by a power of 2. Binary scale is specified with S or s and an optional add-operator (+ or -). *n* is an integer in the base specified by the *decimal-prefix*; for example:

S*n* or S+*n*	Positive binary exponent
s*n* or s+*n*	Positive binary exponent
S-*n* or s-*n*	Negative binary exponent

Examples (floating-constant for constants):

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |CON       |D'1.5            |; Mixed decimal of the form
|        |          |                 |; n.n
|        |CON       |4.5E+10          |; Single-precision floating
|        |          |                 |; constant of the form
|        |          |                 |; n.nE+n
|        |CON       |4.D+15           |; Double-precision floating
|        |          |                 |; constant of the form n.D+n
|        |CON       |D'1.0E-6         |; Negative floating constant of
|        |          |                 |; the form n.nE-n
|        |CON       |1000e2           |; Single-precision floating
|        |          |                 |; constant of the form nen
|SYM     |=         |1777752d+10      |; Double-precision floating
|        |          |                 |; constant of the form nD+n
```

4.4.1.2 Integer-constant

An integer-constant is evaluated as a 64-bit twos-complement integer. (See the twos-complement integer figure in one of the following appropriate Symbolic Machine Instruction manuals: CRAY-2 Computer System Functional Description, publication HR-2000 or Symbolic Machine Instructions Reference Manual, CRI publication SR-0085.) The integer-constant is defined as follows:

integer-constant ::= *base-integer* [*binary-scale base-integer*] |
octal-prefix octal-integer [*binary-scale octal-integer*] |
decimal-prefix decimal-integer [*binary-scale decimal-integer*] |
hex-prefix hex-integer [*binary-scale hex-integer*] .

base-integer
A string of decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) of any length

binary-scale

The *integer* and/or *fraction* is to be multiplied by a power of 2. Binary scale is specified with S or s and an optional add-operator (+ or -). *n* is an integer in the base specified by the *decimal-prefix;* for example:

S*n* or S+*n*	Positive binary exponent
s*n* or s+*n*	Positive binary exponent
s-*n* or S-*n*	Negative binary exponent

base-integer or *octal-prefix* or *decimal-prefix* or *hex-prefix*

Numeric base used for the integer. If no prefix is used, base-integer is determined by the default mode of the assembler or by the BASE pseudo instruction. A prefix can be one of the following:

D' or d'	Decimal (default mode)
O' or o'	Octal
X' or x'	Hexadecimal

octal-integer

A string of octal integers (0, 1, 2, 3, 4, 5, 6, 7) of any length

decimal-integer

A string of decimal integers (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) of any length

hex-integer

A string of hex integers (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A or a, B or b, C or c, D or d, E or e, F or f) of any length

Example (integer-constant for constants):

```
|Location|Result     |Operand        |Comment
|1       |10         |20             |35
|        |           |               |
|        |S1         |O'1234567      |; Octal-prefix followed by
|        |           |               |; octal-integer
|        |A4         |D'50           |; Integer-constant of the form
|        |           |               |; decimal-prefix followed by
|        |           |               |; decimal-integer
|SYM     |=          |h'ffffffa      |; Integer-constant of the form
|        |           |               |; hex-prefix followed by
|        |           |               |; hex-integer
```

4.4.1.3 Character-constants

The character-constant is defined as follows:

```
character-constant ::=  [ character-prefix ] character-string
                        [ character-suffix ] .
```

character-prefix
Character set used for stored constant:

A or a	ASCII character set (default)
C or c	Control Data Display Code
E or e	EBCDIC character set

character-string
Default is a string of zero or more characters from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate a single apostrophe.

character-suffix
Justification and fill of character string:

H or h	Left-justified, blank-filled (default)
L or l	Left-justified, zero-filled
R or r	Right-justified, zero-filled
Z or z	Left-justified, zero-filled, at least one trailing binary zero character guaranteed

Example (character-constant):

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|        |S3        |'*'R          |; ASCII character set (default);
|        |          |              |; right justified, zero filled
|        |CON       |A'ABC'L       |; ASCII character set; left
|        |          |              |; justified, zero filled
|        |S1        |E'XYZ'H       |; EBCDIC character set; left
|        |          |              |; justified, blank filled
|        |CON       |C'OUT'        |; CDC character set; left
|        |          |              |; justified, blank filled;
|        |          |              |; (default)
|        |VWD       |32/'EFG'      |; ASCII character set; left
|        |          |              |; justified, blank filled within
|        |          |              |; a 32-bit field (all default)
```

4.4.2 DATA ITEMS

A character or data item can be used in the operand field of the DATA pseudo instruction and in literals. The length of the data field occupied by a data item is determined by its type and size. The data item is defined as follows:

```
data-item ::= floating-data | integer-data | character-data .
```

4.4.2.1 Floating-data item

A floating-data item occupies one word if single precision and two words if double precision. Floating-point data is defined as follows:

```
floating-data ::= [ sign ] floating-constant .
```

sign Data item is to be stored ones or twos complemented or uncomplemented:

+ or omitted	Uncomplemented
-	Negated (twos complemented)
#	Ones complemented. Although syntactically correct, # is not permitted; a semantic error is generated with floating-data.

The floating-constant is defined as follows:

```
floating-constant ::=  [ decimal-prefix ] floating-decimal
                       [ binary-scale decimal-integer ] ] .
```

The syntax for floating-data is the same as the syntax for floating-constants. See subsection 4.4.1.1, Floating-constant, for a description of floating constants.

Example (floating-constant for data items):

```
|Location|Result    |Operand        |Comment
|1       |10        |20             |35
|        |          |               |
|        |DATA      |D'1345.567     |; Decimal floating data item of
|        |          |               |; the form n.n
|        |DATA      |1345.E+1       |; Decimal floating data item of
|        |          |               |; the form n.E+n
|        |DATA      |D'1.5          |; Decimal of the form n.n
|        |DATA      |4.5E+10        |; Single-precision floating
|        |          |               |; constant of the form
|        |          |               |; n.nE+n
```

Examples (continued):

```
|Location|Result    |Operand        |Comment
|1       |10        |20             |35
|        |          |               |
|        |DATA      |4.D+15         |; Double-precision floating
|        |          |               |; constant of the form n.D+n
|        |DATA      |D'1.0E-6       |; Negative floating constant of
|        |          |               |; the form n.nE-n
|        |DATA      |1000e2         |; Single-precision floating
|        |          |               |; constant of the form nen
|        |DATA      |1.5S2          |; Floating binary scale data
|        |          |               |; item of the form n.nSn
```

4.4.2.2 Integer-data item

An integer-data item occupies one 64-bit word and is defined as follows:

integer-data ::= [*sign*] *integer-constant* .

sign Data item is to be stored ones or twos complemented or uncomplemented:

+ or omitted	Uncomplemented
-	Negated (twos complemented)
#	Ones complemented

```
integer-constant ::=
          base-integer [ binary-scale base-integer ] |
          octal-prefix octal-integer
          [ binary-scale octal-integer ] |
          decimal-prefix decimal-integer
          [ binary-scale decimal-integer ] |
          hex-prefix hex-integer [ binary-scale hex-integer ] .
```

The syntax for the integer-data is the same as the syntax for the integer-constant. See subsection 4.4.1.2, Integer-constant, for a detailed description of integer-constants.

Example (integer-constant for data):

```
|Location|Result    |Operand        |Comment
|1       |10        |20             |35
|        |          |               |
|        |DATA      |+O'20          |; Octal-integer
|        |          |               |
|        |VWD       |40/0,24/O'200  |
```

4.4.2.3 Character-data item

The character-data item is as follows:

```
character-data ::= [ character-prefix ] character-string
                   [ character-count ] [ character suffix ] .
```

character-prefix
Character set used for stored constant:

A or a	ASCII character set (default)
C or c	Control Data Display Code
E or e	EBCDIC character set

character-string
Default is a string of zero or more characters from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate a single apostrophe.

character-count
Length of the field, in number of characters, into which the data item is to be placed. If *count* is not supplied, the length is the number of words needed to hold the character string. If a count field is present, the length is the character count times the character width, so length is not necessarily an integral number of words. The character width is 8 bits for ASCII or EBCDIC, 6 bits for Control Data Display Code.

If an asterisk is in the count field, then the actual number of characters in the string is used as the count. The case where two apostrophes are used to represent a single apostrophe is counted as a single character.

If the base is M (mixed), CAL assumes that count is decimal. Refer to section 4, Cray Assembly Language, for a description of mixed base.

character-suffix
Justification and fill of character string:

H or h	Left-justified, blank-filled (default)
L or l	Left-justified, zero-filled
R or r	Right-justified, zero-filled
Z or z	Left-justified, zero-filled, at least one trailing zero character guaranteed

Example (character-data):

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |DATA      |A'ERROR IN DSN'  |; ASCII character set with
|        |          |                 |; left justification and blank
|        |          |                 |; fill by default; stored in two
|        |          |                 |; words.
|        |DATA      |E'error in dsn'R; EBCDIC character set; right
|        |          |                 |; justified, zero filled; stored
|        |          |                 |; in two words.
|        |DATA      |'Error'          |; Default ASCII Character set
|        |          |                 |; left justified and blank
|        |          |                 |; filled by default; stored in
|        |          |                 |; one word.
|        |DATA      |'Error'*         |; Default ASCII character set;
|        |          |                 |; that is stored in 5 character
|        |          |                 |; positions (40 (5*8) bits)
```

4.4.3 LITERALS

Literals are read-only data items whose storage is controlled by CAL. Specifying a literal allows you to implicitly insert a constant value into memory. The actual storage of the literal value is the responsibility of the assembler. Literals can only be used in expressions, because the address of a literal, rather than its value is used.

The first use of a literal value in an expression causes the assembler to store the data item in one or more words in a special, local memory block known as the literals section. Subsequent references to a literal value, do not produce multiple copies of the same literal.

Since literals can map into the same location in the literals section, CAL checks for the presence of a matching literal in the literals section before new entries are added to the section. This check is made bit by bit. If the current string identically matches any string currently stored in the literals section, CAL maps that string to the location of the matching string. If the current string does not identically match any of the strings currently stored in the literals section, the current string is considered to be unique and is assigned a location in the literals section.

The following special syntaxes are in effect for literals:

- Literals always have the following attributes:
  - Relocatable (relative)
  - Word (address)
- Literals cannot be specified as character strings of zero bits. The actual constant within a literal must have a bit length greater than zero. In actual use, you must specify at least one 6- or 8-bit character as follows:
  - 6 bits for CDC character set
  - 8 bits for ASCII character set (default)
  - 8 bits for EBCDIC character set
- By default, literals always come out on full-word boundaries. Trailing blanks are added to fill the word to the next word boundary. The following special characters can be specified with literals:

A literal is defined as follows when used as an element of an expression.

literal ::= "=" *data-item* .

Data-item is defined as follows:

data-item ::= floating-data | integer-data | character-data .

The syntax of the data-item for literals is the same as the syntax for data-items for constants. For a complete description of the data item, see Data-item in this section.

Examples:

1. Literals can be specified with single- or double-precision; the default is single-precision. Single-precision literals are stored in 1 64-bit word. Double-precision literals are stored in 2 64-bit words.

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|        |CON       |=1.5         |; Single-precision literal
|        |CON       |=1.5D1       |; Double-precision literal
```

2. The following example illustrates how the the ASCII character a is stored when ='a'H is specified; ^ represents a blank character. If ='a' is specified, this same value is generated.

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|        |CON       |='a'H         |; ASCII character set by default
|        |          |              |; left justify blank fill
```

01100001	^^^^^^^^	^^^^^^^^	^^^^^^^^	^^^^^^^^	^^^^^^^^	^^^^^^^^	^^^^^^^^

Diagram 4-1. ASCII Character with Left Justification and Blank Fill

3. The following examples illustrates how the the ASCII character a is stored.

   a. In the following example, ='a'L is specified; specifying ='a'R or ='a'Z generates the same value.

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|        |CON       |='a'L         |; ASCII character set by default
|        |          |              |; left justify zero filled
```

01100001	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Diagram 4-2. ASCII Character with Left Justification and Zero Fill

   b. The following example illustrates how the the ASCII character a is stored when ='a'R is specified.

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|        |CON       |='a'R        |; ASCII character set by default
|        |          |             |; right-justified, zero-filled
```

00000000	00000000	00000000	00000000	00000000	00000000	00000000	01100001

Diagram 4-3. ASCII Character with Right Justification and Zero Fill

c. The following example illustrates how the the ASCII character a is stored when ='a'*R is specified. The value is right-justified in the first 8 bits of word 4.

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|        |CON       |'a'*R        |; ASCII character set by default
|        |          |             |; right-justified, zero-filled
```

01100001	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Diagram 4-4. ASCII Character with Right Justification in 8 Bits

4. The following example illustrates the declaration of the three character sets available to CAL.

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|        |CON       |='A'         |; 8-bit ASCII character
|        |CON       |=A'A'        |; 8-bit ASCII character;
|        |CON       |=C'A'        |; 6-bit CDC character
|        |CON       |=E'A'        |; 8-bit EBCDIC character
```

5. The following examples illustrate how literals can be specified using H, L, R, Z:

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|        |CON       |='AB'3       |; Left justified with one blank
|        |          |             |; padded on the right; default
|        |CON       |='AB'3H      |; Left justified with one blank
|        |          |             |; padded on the right; default
|        |CON       |='AB'6R      |; Right justified, filled with
|        |          |             |; four leading zeros
|        |CON       |='AB'6Z      |; Left justified, padded with
|        |          |             |; four trailing zeros
```

4.5 SPECIAL ELEMENTS

Special elements are used to obtain the current value of the location counter, the origin counter, the word pointer, and the parcel pointer. Special elements can occur as elements of expressions. Expression elements are described in subsection 4.7, Expressions. The origin, location, word-bit-position, and parcel-bit-position counters are described in section 3, The CAL Program. Special elements are defined as follows:

```
special element ::= "*" | "*A" | "*a" | "*B" | "*b" | "*O" | "*o" |
                    "*P" | "*p" | "*W" | "*w" .
```

Special elements have the following special meanings to the assembler.

Element	Description
*	Location counter; denotes a value equal to the current value of the location counter with parcel-address attribute and absolute, immobile, or relocatable attributes. The location counter is absolute if it has been modified by the LOC pseudo using an expression that has a relative attribute of absolute. The location counter is immobile if it is relative to either a STACK section or a TASKCOM section. The location counter is relocatable in all other cases.
*A or *a	Absolute location counter; denotes a value equal to the current value of the location counter with parcel-address and absolute attributes.

Element	Description
*O or *o	Origin counter; denotes a value equal to the current value of the origin counter relative to the beginning of the current section. The origin counter has an address attribute of parcel. If the current section is a section with a type of STACK or TASKCOM, it has an immobile attribute. In all other cases it has a relative attribute of relocatable.
*B or *b	Absolute origin counter; denotes a value equal to the current value of the origin counter relative to the beginning of the section with parcel-address and absolute attributes.
*W or *w	Word pointer; denotes a value equal to the current value of the word-bit-position counter with absolute and value attributes. *W is relative to the word and the word-bit-position counter is almost always equal to 0, 16, 32, or 48. CAL issues a warning message when the word-bit-position counter has a value other than 0 (not pointing at a word boundary) and is used in an expression.
*P or *p	Parcel pointer; denotes a value equal to the current value of the parcel-bit-position counter with absolute and value attributes. The range of possible values for *P is 0 through 15. CAL issues a warning message when the parcel-bit-position counter has a value other than 0 (not pointing at a parcel boundary) and is used in an expression. The following statement defines where you are within a parcel and is almost always 0: SYM1 = *P

4.6 ELEMENT PREFIXES FOR SYMBOLS, CONSTANTS, OR SPECIAL ELEMENTS

Element prefixes have the following form:

```
element-prefix ::= "P." | "p." | "W." | "w." .
```

A symbol, constant, or special element can be prefixed by an element-prefix (P. or p. for parcel or W. or w. for word) causing the value to assume an attribute of parcel address or word address, respectively, in the expression in which the reference appears.

A prefix does not permanently alter the attribute of a symbol; the effect of a prefix is for the current reference only.

4.6.1 P. - PARCEL-ADDRESS PREFIX

A symbol, special element, or constant can be prefixed by P. or p. to specify the attribute of parcel address. If a symbol (*sym*) has the attribute of word address, the value of P.*sym* or p.*sym* is the value of *sym* multiplied by four. Each Cray word is divided into four parcels that are designated as a, b, c, and d. Each parcel has a 2-bit value associated with it; 00_2 for a, 01_2 for b, 10_2 for c, and 11_2 for d. To find the exact parcel that is being addressed, multiply the word address by four. For example, the following word address attributes are translated into parcel address attributes:

Word	Equation	Value	Parcel Representation
2	2X4	O'10	2a
4	4X4	O'20	4a
0	0X4	O'0	0a

A P. or p. that is specified for an element with value address attribute does not cause the value to be divided by four. The value can, however, be used to assign the parcel address attribute to the element.

A P. or p. that is specified for an element with parcel address attribute does not alter its characteristics.

Figure 4-1 illustrates the numbering of parcels a, b, c, and d in a 6-word block.

	Parcel a	Parcel b	Parcel c	Parcel d
Word 0	0	1	2	3
Word 1	4	5	6	7
Word 2	10	11	12	13
Word 3	14	15	16	17
Word 4	20	21	22	23
Word 5	24	25	26	27

Figure 4-1. Word-parcel Conversion for Six Words

An expession is defined as follows:

```
expression ::= embedded-argument |
               [ add-operator ] term { add-operator term } .
```

embedded-argument
An embedded-argument can be any argument-character that is enclosed in parentheses. If parentheses are used with an embedded argument, each open parenthesis must have a matching close parenthesis. For example:

```
(3*SYMBOL+2-(2*SYMBOL))
```

4.7.1 ADD-OPERATOR

An add-operator joins two terms in an expression or precedes the first term of an expression. Add-operators are defined as follows:

```
add-operator ::= "+" | "-" .
```

4.7.2 TERMS

A term consists of one or more prefixed-elements joined by special characters referred to as multiply-operators. The multiply-operators complete all multiplication and division before the add-operators complete addition or subtraction. The following general rules apply for terms.

- Only one prefixed-element within a term can have a relative attribute of immobile or relocatable. All other prefixed-elements, if any, in that term must have relative attributes of absolute.
- A prefixed-element with a relative attribute of external, if present, must be the only prefixed-element of the term. If preceded by an adding operator, that operator must be a +.
- The prefixed-element to the right of / must have a relative attribute of absolute.
- A term containing / must have an attribute of absolute up to the point at which the / is encountered (see the description of term attributes).
- Division by 0 produces an error.

4.6.1 P. - PARCEL-ADDRESS PREFIX

A symbol, special element, or constant can be prefixed by P. or p. to specify the attribute of parcel address. If a symbol (*sym*) has the attribute of word address, the value of P.*sym* or p.*sym* is the value of *sym* multiplied by four. Each Cray word is divided into four parcels that are designated as a, b, c, and d. Each parcel has a 2-bit value associated with it; 00_2 for a, 01_2 for b, 10_2 for c, and 11_2 for d. To find the exact parcel that is being addressed, multiply the word address by four. For example, the following word address attributes are translated into parcel address attributes:

Word	Equation	Value	Parcel Representation
2	2X4	O'10	2a
4	4X4	O'20	4a
0	0X4	O'0	0a

A P. or p. that is specified for an element with value address attribute does not cause the value to be divided by four. The value can, however, be used to assign the parcel address attribute to the element.

A P. or p. that is specified for an element with parcel address attribute does not alter its characteristics.

Figure 4-1 illustrates the numbering of parcels a, b, c, and d in a 6-word block.

	Parcel a	Parcel b	Parcel c	Parcel d
Word 0	0	1	2	3
Word 1	4	5	6	7
Word 2	10	11	12	13
Word 3	14	15	16	17
Word 4	20	21	22	23
Word 5	24	25	26	27

Figure 4-1. Word-parcel Conversion for Six Words

Example:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|SYM1    |=         |*           |; SYM1 is equal to the location
|        |          |            |; counter with parcel and
|        |          |            |; relocatable attributes.
|        |S1        |SYM1        |; Register S1 gets the
|        |          |            |; relocatable parcel address of
|        |          |            |; SYM1.
|        |S1        |P.SYM1      |; The same value that was
|        |          |            |; generated by the last
|        |          |            |; statement is produced.
```

4.6.2 W. - WORD-ADDRESS PREFIX

A symbol, special element, or constant can be prefixed by W. or w. to specify the attribute of word address. If a symbol (*sym*) has the attribute of parcel address, the value of W.*sym* or w.*sym* is the value of *sym* divided by four. When converting from parcel address attribute to a word address attribute, divide the parcel address by 4. When the conversion is completed, the result is always understood to be pointing at parcel a.

If the parcel address is not pointing at a word boundary, CAL issues a warning message and truncates the division to a word boundary. For example, the following parcel address attributes are converted into word-address attributes:

Parcel Representation	Value	Equation	Word	Truncation Warning
0c	2	2/4	0	Yes
3a	14	14/4	3	No
5c	26	26/4	5	Yes
0a	0	0/4	0	No
6a	30	30/4	6	No

A W. or w. prefix specified for an element with a value-address attribute does not cause the value to be divided by four. The value can, however, be used to assign the word-address attribute to the element.

A W. or w. prefix specified for an element with a word-address attribute does not alter its characteristics.

Example (word-address prefix):

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|SYM2    |=         |W.*           |; Word and relocatable
|        |          |              |; attributes
|        |A0        |W.ADDR        |
|        |          |              |
|        |A4        |W.BUFF+O'100  |
```

4.7 EXPRESSIONS

The result and operand fields for many source statements consist of expressions. An expression consists of one or more terms joined by special characters referred to as adding operators (add-operators in the BNF). Figure 4-2 is a diagram of an expression. A term consists of one or more special elements, constants, symbols, or literals (prefixed-element in the BNF) joined by multiplying operators (multiply-operator in the BNF). Figure 4-3 is a diagram of a term.

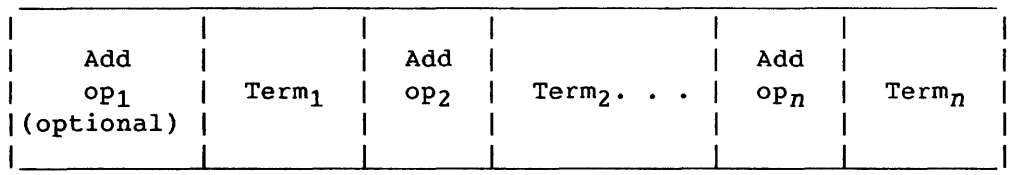

Figure 4-2. Diagram of An Expression

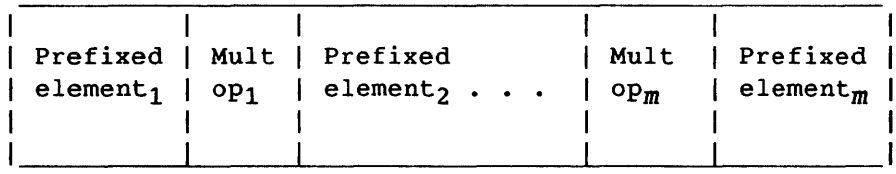

Figure 4-3. Diagram of A Term

An expression is defined as follows:

```
expression ::= embedded-argument |
               [ add-operator ] term { add-operator term } .
```

embedded-argument
An embedded-argument can be any argument-character that is enclosed in parentheses. If parentheses are used with an embedded argument, each open parenthesis must have a matching close parenthesis. For example:

```
(3*SYMBOL+2-(2*SYMBOL))
```

4.7.1 ADD-OPERATOR

An add-operator joins two terms in an expression or precedes the first term of an expression. Add-operators are defined as follows:

```
add-operator ::= "+" | "-" .
```

4.7.2 TERMS

A term consists of one or more prefixed-elements joined by special characters referred to as multiply-operators. The multiply-operators complete all multiplication and division before the add-operators complete addition or subtraction. The following general rules apply for terms.

- Only one prefixed-element within a term can have a relative attribute of immobile or relocatable. All other prefixed-elements, if any, in that term must have relative attributes of absolute.
- A prefixed-element with a relative attribute of external, if present, must be the only prefixed-element of the term. If preceded by an adding operator, that operator must be a +.
- The prefixed-element to the right of / must have a relative attribute of absolute.
- A term containing / must have an attribute of absolute up to the point at which the / is encountered (see the description of term attributes).
- Division by 0 produces an error.

Example:

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|SYM     |=         |*            |; Relocatable and parcel
|        |          |             |; attributes.
|        |S1        |SYM*1        |; One term within an expression.
|        |S2        |SYM*1+1      |; Two terms within an
|        |          |             |; expression.
|        |S3        |1*2*3/4      |; Every prefixed-element
|        |          |             |; preceding a / must have the
|        |          |             |; attribute of absolute and the
|        |          |             |; prefixed-element following the
|        |          |             |; / must have an attribute of
|        |          |             |; absolute.
```

A term is defined as follows:

term ::= *prefixed-element* { *multiply-operator prefixed-element* } .

prefixed-element ::= ["#"] [*element-prefix*] *element* .

multiply-operator ::= "*" | "/" .

Examples (terms):

Term	Description
SIGMA*5	Two elements, SIGMA (symbol) and 5 (constant) are joined by a multiplying operator (*).
DELTA	A single-element term

4.7.2.1 Prefixed-elements

A prefixed-element is defined as follows:

prefixed-element ::= [#] [element-prefix] element .

Complement character (#) - If an element is prefixed with the complement character (#), the element itself must have a relative attribute of absolute.

Element-prefix - If an element is prefixed with an element-prefix, the attribute of the element is as follows:

P. or p. Parcel-address attributes
W. or w. Word-address attributes

See subsection 4.6, Element Prefixes for Symblols, Constants, or Special Elements, for information about element-prefixes.

Elements - An element can be a special element, constant, symbol, or literal. Elements can be optionally preceded by a complement character (#) or an element prefix (P. or W.). Elements are defined as follows:

element ::= *special-element* | *constant* | *symbol* | *literal* .

element For more detailed information about *element*, see subsection 4.5, Special Elements, for *special-element*, subsection 4.4.1, Constants, for *constant*, subsection 4.3, Symbols, for *symbol*, and subsection 4.4.2, Data Items, for *literal*.

Examples (elements):

Element	Description
SIGMA	Symbol
*	Special element
*W	Special element
O'77S3	Numeric constant
A'ABC'R	Character constant
=A'ABC'	Literal

4.7.2.2 Multiply-operator

A multiply-operator joins two prefixed-elements. Multiply-operators are defined as follows:

multiply-operator ::= "*" | "/" .

4.7.2.3 Term attributes

Every prefixed-element in a term has a relative and an address attribute associated with it. CAL assigns relative and address attributes to the entire term by evaluating each prefixed-element in the term.

The relative and address attributes for a term vary as CAL evaluates each prefixed-element in the term. The term's final attribute is the attribute in effect when the final (right-most) element of the term is evaluated. As CAL encounters each prefixed-element in the left-to-right scan of a term, it assigns an attribute to the term based on the multiply-operator (if any) preceding the prefixed-element, the attribute of any previous partial term, and the attribute of the prefixed-element currently being evaluated.

Relative attributes - The prefixed-elements and multiply-operators comprising a term determine the term's relative attributes. CAL assigns every term a relative attribute determined by the following rules:

Rule	Attribute	Description
1	Absolute	A term assumes the attribute of absolute if every prefixed-element is absolute.
2	Immobile	A term assumes an attribute of immobile if it contains one prefixed-element with immobile attributes, zero or more prefixed-elements with absolute attributes, and no prefixed-elements with relocatable or external attributes. Thus an immobile term can contain one immobile prefixed-element with the remaining prefixed-elements being absolute.
3	Relocatable	A term assumes an attribute of relocatable if it contains one prefixed-element with relocatable attributes, zero or more prefixed-elements with absolute attributes, and no prefixed-elements with immobile or external attributes. Thus a relocatable term can contain one relocatable prefixed-element with the remaining prefixed-elements being absolute.
4	External	A term assumes the attribute of external if it consists of one prefixed-element and the prefixed-element is external.

Examples:

Term	Evaluation
2*4/3*4	Absolute (2) * absolute (4) is evaluated as absolute. Absolute (2*4) / absolute (3) is evaluated as absolute. Absolute (2*4/3) * absolute (4) is evaluated as absolute; rule 1.

Term	Evaluation
STKSYM*3	Immobile (STKSYM) * absolute (3) is evaluated as immobile; rule 2.
2*SYM1*2	Absolute (2) * relocatable (SYM1) is evaluated as relocatable. Relocatable (2*SYM1) * absolute (2) is evaluated as relocatable; rule 3.
EXT1	One external (EXT1) element is evaluated as external; rule 4.
EXT2*SYM1	External (EXT2) * relocatable (SYM1) produces an error; rule 4.
4*SYM1/4	Absolute (4) * relocatable (SYM1) is evaluated as relocatable; relocatable (4*SYM1) / 4 produces an error. All prefixed-elements to the left of the / must have a relative attribute of absolute; see general rules for terms in subsection 4.7.2, Terms.

Address attributes - CAL assigns every term one of the following address attributes:

- Parcel-address
- Word-address
- Value

Figure 4-4 indicates how address attributes are assigned to terms and partial terms. *Pterm, Wterm,* and *Vterm* denote the attribute of the partial term resulting from all elements evaluated before the current element. In figure 4-4, P, W, and V denote an element being incorporated into the term and having an attribute of parcel-address, word-address, or value, respectively.

If a partial term has the address attribute of the left column and is multiplied or divided by a prefixed-element with the address attribute of the top horizontal row, the resulting attribute is determined at the intersection of the column and row by the arithmetic operator position in the upper left corner of table.

The results for multiplication and division are given in the top (*) and bottom (/) halves of each box on the chart, respectively. For example, if partial term Vterm is multiplied by a prefixed-element with an address attribute of word, the address attribute for the new partial term is word.

A two-digit value following an address attribute indicates that although a result is specified, CAL issues a warning message that corresponds to the two-digit superscript. For example, if the partial term Vterm is divided by a prefixed element with an address attribute of parcel, the result is value and message 84 is issued:

```
Partial term with value address is divided by parcel element
```

See appendix D, Diagnostic Messages, for the text that is associated with messages 80 through 87.

Partial Term \ 2nd Term	* /	V	P	W
Vterm		V / V	P / V 84	W / V 86
Pterm		P / P	P 80 / V	V 82 / V 87
Wterm		W / W	V 81 / V 85	V 83 / V

V - Value
P - Parcel
W - Word
nn - Warning message number

Figure 4-4. Address Attribute Assignment Chart

4.8 EXPRESSION EVALUATION

Expressions are evaluated from left to right. Each term is evaluated from left to right with CAL performing 64-bit integer multiplication or division as each multiply-operator is encountered. Expressions are defined as follows:

```
expression ::= embedded-argument |
               [ add-operator ] term { add-operator term } .
```

NOTE

The embedded-argument is intended for use with macros and opdefs and should not be included in expressions. Although the embedded-argument is syntactically correct, the CAL expression evaluator cannot evaluate expressions that contain embedded-arguments. For example:

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|sym1    |=         |1             |; Valid expression.
|sym2    |=         |(1)           |; Syntactically
|        |          |              |; correct, but CAL
|        |          |              |; issues error
|        |          |              |; message.
```

When a complete term has been evaluated, it is added or subtracted from the sum of the previous terms. CAL does not check for overflow and underflow.

The assembler treats each element as a 64-bit twos-complement integer. Character constants are left- or right-justified within a field width equal to the destination field. If the field width is shorter than the length of the character constant a warning message is issued. Complemented elements are complemented in the right-most bits in a field width equal to the destination field.

NOTE

CAL processes floating-constants as expected when they are specified as single uncomplemented prefixed-elements within an expression. If floating-constants are used in any other way, an appropriate warning message is issued and integer arithmetic is used to evaluate the expression. CAL processes the floating-constants within the expressions of the following examples as expected.

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|A       |CON       |1.0         |
|B       |CON       |-1.0        |
|C       |CON       |4.5         |
|D       |CON       |.3          |
|E       |CON       |-.75        |
```

CAL issues an appropriate warning message and evaluates the floating-constants within the expressions of the following examples using integer arithmetic:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|G       |CON       |1.0+2.0     |
|H       |CON       |-1*3.4      |
|I       |CON       |-#1.0       |
```

Examples:

1. The following example demonstrates how the result of a VWD with a nine-bit destination field is stored; ^ represents a blank space.

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |VWD       |D'9/'abc'+1 |; The terms of the expression
|        |          |            |; 'abc' and 1
```

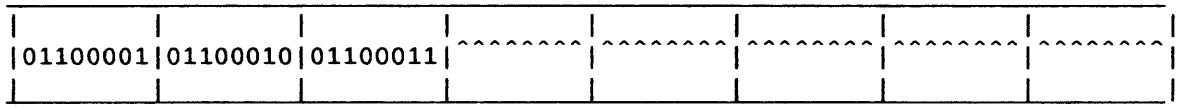

Diagram 4-5. 64-bit ASCII Representation of 'abc', Left Justified

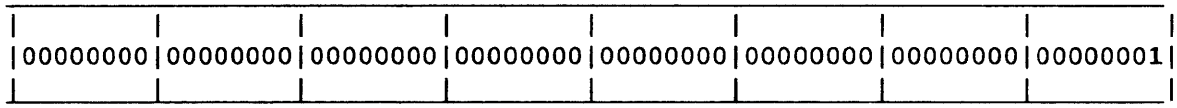

Diagram 4-6. 64-bit Representation of 1

Diagrams 4-5 and 4-6 contain the ASCII representations of the character strings 'abc' and 1, respectively. Since the character constant is left-justified by default within a field width equal to the 9 bits specified in the example, the 64-bit representation of 'abc' is actually as follows:

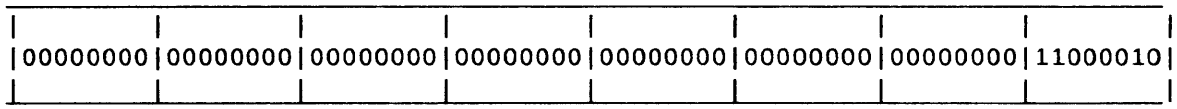

Diagram 4-7. ASCII Representation of 'abc', Left Justified in 9 Bits

CAL adds the value 1 (diagram 4-6) to the value shown in diagram 4-7 (011000010), and stores it in the destination field (diagram 4-8). CAL issues a warning message stating that the character string 'abc' has been truncated. The destination field contains a value of 303 (011000011).

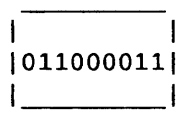

Diagram 4-8. Result of VWD with 9-bit Destination Field

2. The following example demonstrates that complemented elements are complemented in the right-most bits of a field width equal to the destination field.

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|        |VWD       |D'4/#1+1     |; The terms of the expression
|        |          |             |; are the complement of 1 and
|        |          |             |; the value 1.  The destination
|        |          |             |; field is 4 bits wide.
```

11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111110

Diagram 4-9. 64-bit Representation of the Complement of 1

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000001

Diagram 4-10. 64-bit Representation of 1

Diagrams 4-9 and 4-10 contain the complement of 1 and the ASCII representation for the value 1 (0001), respectively. Diagram 4-11 shows the actual value of the complement of 1 is stored in the right-most bits of a word in memory.

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00001110

Diagram 4-11. Complement of 1 Stored in the Right-most Bits of a 4-bit Field

The character string 1110 (diagarm 4-11) is stored in the destination field, CAL adds the value 1 to the destination field, and the result (1111) is stored as shown in diagram 4-12.

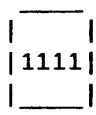

Diagram 4-12. Result of VWD with 4-bit Destination Field

4.8.1 EVALUATING IMMOBILE AND RELOCATABLE TERMS WITH COEFFICIENTS

An immobile term has one immobile prefixed-element, no relocatable or external prefixed-elements, and zero or more absolute prefixed-elements. A relocatable term has one relocatable prefixed-element, no immobile or external prefixed-elements, and zero or more absolute prefixed-elements.

An immobile term has a 64-bit integer coefficient associated with it, equal to the value of the term obtained when a 1 is substituted for the immobile element. The value of an immobile term is the value of the immobile element multiplied by the coefficient.

A relocatable term has a 64-bit integer coefficient associated with it, equal to the value of the term obtained when a 1 is substituted for the relocatable element. The value of a relocatable term is the value of the relocatable element multiplied by the coefficient.

Every section has two relative section coefficients, one representing an immobile relative attribute and one representing a relocatable relative attribute. These relative section coefficients are initialized to zero before the evaluation of each expression. As each term is evaluated within an expression, the term's coefficient is either added to or subtracted from the corresponding coefficient of the corresponding section depending on the sign immediately preceding the term. When each term within an expression has been evaluated, the expression is assigned a relative attribute as follows:

- Absolute; if the expression contains no external terms and all of the coefficients for all of the sections are zero.

- Immobile; if the expression contains no external terms and all of the coefficients for all of the sections are zero except for one immobile coefficient that must have a value of 1. The expression is immobile relative to the section with the coefficient of one.

- Relocatable; if the expression contains no external terms and all of the coefficients for all of the sections are zero except for one relocatable coefficient that must have a value of 1. The expression is relocatable relative to the section with the coefficient of one.

- External; if the expression contains one external term and all of the coefficients for all of the sections are zero.

- Invalid; all other cases.

If, for example, SYMBOL is assumed to be relocatable, SYMBOL*2+1-SYMBOL is considered a valid expression when it is evaluated by CAL. Since SYMBOL is relocatable, substituting one for SYMBOL generates three terms (1*2, +1, and -1). The first term (1*2) includes the relocatable term SYMBOL. A value of 2 is stored with the coefficient maintained by CAL for the relocatable section to which SYMBOL is relative. The second term (+1) is absolute and does not effect the evaluation of the relocatable coefficient. The third term (-1) includes the relocatable term SYMBOL. A one is subtracted from the coefficient maintained by CAL for the relocatable section named SYMBOL.

When the entire term is evaluated, the coefficient associated with the relocatable term SYMBOL equals one. Since all of the relocatable terms within the expression are relative to a single section and the section's final coefficient is one, the expression is relocatable relative to that section.

Every relocatable symbol is relative to some section. All sections have an initial coefficient of zero before expression evaluation. The operator immediately preceding a relocatable term is the operator associated with that term. For example, the coefficient for SYMBOL is maintained as -1. When the sign of a coefficient is not indicated, it is assumed to be positive. The coefficient for SYMBOL*1 is maintained as +1*1. If 1a (100) is substituted for SYMBOL in the following expression:

```
SYMBOL*2+1-SYMBOL
```

the binary to be evaluated is 100*010+001-100. CAL evaluates the string from left to right. The following partial results are obtained:

```
100*010=1000
1000+0001=1001
1001-0100=0101=1b
```

The final result (1b) is the result that we would expect to be generated. The following example demonstrates the correct and incorrect use of a relocatable term.

Example:

```
|Location|Result    |Operand                |Comment
|1       |10        |20                     |35
|        |          |                       |
|        |IDENT     |                       |
|SYMBOL  |=         |*                      |; SYMBOL is given a value equal
|        |          |                       |; to the current location
|        |          |                       |; counter.
|        |S1        |SYMBOL*2+1-SYMBOL      |; When evaluated, this
|        |          |                       |; expression produces a value
|        |          |                       |; equal to the current location
|        |          |                       |; counter plus 1.  The value is
|        |          |                       |; relocatable.
|        |S1        |SYMBOL*2+1             |; When evaluated, this
|        |          |                       |; expression produces a value
|        |          |                       |; equal to twice the current
|        |          |                       |; location counter plus 1.  The
|        |          |                       |; value is not relocatable.
|        |          |                       |; CAL produces an error message.
|        |END       |                       |
```

The term SYMBOL*2+1 is not relocatable because the results generated are dependent on the location where the loader puts the module. If the loader puts the module at 400, SYMBOL*2+1=801. If the loader puts the module at 200, SYMBOL*2+1=401. If a term is evaluated and found to be not relocatable, CAL issues a message with a priority of error.

Example (relocatable):

```
|Location|Result    |Operand                |Comment
|1       |10        |20                     |35
|        |          |                       |
|        |IDENT     |TEST                   |
|SNAME1  |SECTION   |                       |
|SYMBOL1 |BSS       |4                      |
|SYMBOL2 |=         |W.*                    |
|        |BSS       |5                      |
|        |SECTION   |*                      |
|SNAME2  |SECTION   |                       |
|SYMBOL3 |BSS       |3                      |
|        |SECTION   |*                      |
|SYMBOL4 |=         |3*SYMBOL2+SYMBOL3-1-SYMBOL2-2*SYMBOL1
|        |END       |                       |
```

The expression 3*SYMBOL2+SYMBOL3-1-SYMBOL2-2*SYMBOL1 contains five terms, four of which are relocatable, and is evaluated as follows:

Term	Value of Coefficient	Attribute
3*SYMBOL2	3*1	Relocatable (relative to SNAME1)
+SYMBOL3	+1	Relocatable (relative to SNAME2)
-1		Absolute
-SYMBOL2	-1	Relocatable (relative to SNAME1)
-2*SYMBOL1	-2*1	Relocatable (relative to SNAME1)

In the previous example, the coefficients for the sections SNAME1, and SNAME2 were initialized to zero before the expression was evaluated. The main section has a coefficient of zero. When the coefficients for the relocatable terms relative to SNAME1 are evaluated, the result is zero (+3-1-2). When the coefficients for the relocatable terms for SNAME2 are evaluated, the result (+1) is 1.

SYMBOL4 obtains a relative attribute of relocatable because one section in the expression has a coefficient of 1 (SNAME2) and all other sections (SNAME1) maintained for the expression have coefficients of 0. The final expression is relocatable relative to SNAME2, because SNAME2 is the section with the coefficient of 1.

The address attribute of the expression is evaluated as follows:

Term	Partial Term	Attribute
3*SYMBOL2	Value*word	Word (see figure 4-4)
+SYMBOL3	Word	Word (see figure 4-4)
-1	Value	Value (see figure 4-4)
-SYMBOL2	Word	Word (see figure 4-4)
-2*SYMBOL1	Value*word	Word (see figure 4-4)

The address attribute for the entire expression is word. For a description of the manner in which parcel-address, word-address, and value attributes are assigned to entire expressions, see subsection 4.9, Expression Attributes.

The value of the expression 3*SYMBOL2+SYMBOL3-1-SYMBOL2-2*SYMBOL1=O'7 and is calculated as follows:

Term	Result	Description
3*SYMBOL2	3*4=O'14	SYMBOL2 begins with word 4 in section SNAME1; 4 is substituted for SYMBOL2.
SYMBOL3	0	SYMBOL3 begins with word 0 in section SNAME2; 0 is substituted for SYMBOL3.
-1	-1	Term 3 is absolute; no substitution.

Term	Result	Description
-SYMBOL2	-4	SYMBOL2 begins with word 4 in section SNAME1; 4 is substituted for SYMBOL2.
-2*SYMBOL1	-2*0=0	SYMBOL1 begins with word 0 in section SNAME1; 0 is substituted for SYMBOL1.

When the values for the terms (O'14+0-1-4-0) are substituted for the expression (3*SYMBOL2+SYMBOL3-1-SYMBOL2-2*SYMBOL1), the result is 7.

Example (immobile):

Location	Result	Operand	Comment
1	10	20	35
	ident	test	
taskc	section	taskcom	
tcsym	bss	4	
	section	*	
symbol	=	taskc+tcsym-taskc	

The expression taskc+tcsym-taskc contains three terms, two that are relocatable and one that is immobile. The expression is evaluated as follows:

Term	Value of Coefficient	Attribute
taskc	+1	Relocatable (relative to taskc)
+tcsym	+1	Immobile (relative to taskc)
-taskc	-1	Relocatable (relative to taskc)

In the previous example, the relative section coefficients for relocatable taskc and immobile tcsym were initialized to zero before the expression was evaluated. When the coefficients for the relocatable terms relative to taskc are evaluated, the result is zero (+1-1=0). When the coefficient for the immobile term (tcsym) is evaluated, the result (+1) is 1. Since the term with the relative attribute of immobile has the coefficient of 1, the entire expression is assigned a relative attribute of immobile.

The address attribute of the expression is evaluated as follows:

Term	Partial Term	Attribute
taskc	Word	Word (see figure 4-4)
+tcsym	Word	Word (see figure 4-4)
-taskc	Word	Word (see figure 4-4)

The address attribute for the entire expression is word. For a description of the manner in which parcel-address, word-address, and value attributes are assigned to entire expressions, see subsection 4.9, Expression Attributes.

The value of the expression taskc+tcsym-taskc is calculated as follows:

Term	Result	Description
taskc	0	taskc is assigned a value of 0 relative to the task common section taskc; 0 is substituted for taskc.
+tcsym	0	tcsym begins with word 0 in taskcom section taskc; 0 is substituted for tcsym.
-taskc	0	taskc is assigned a value of 0 relative to the task common section taskc; 0 is substituted for taskc.

When the values for the terms (0+0-0) are substituted for the expression (taskc+tcsym-taskc), the result is 0.

4.9 EXPRESSION ATTRIBUTES

The expression attributes for a full expression are determined by evaluating the terms within an expression. The assembler can assign the following attributes to an expression:

- Relative
  - Absolute
  - Immobile
  - Relocatable
  - External
- Address
  - Parcel-address
  - Word-address
  - Value

4.9.1 ABSOLUTE, IMMOBILE, RELOCATABLE, or EXTERNAL

Every expression assumes one of the following relative attributes:

- Absolute
- Immobile
- Relocatable
- External

An expression is absolute if no external terms are present and the coefficients of all other sections are 0.

An expression is immobile if the coefficient for every section within the current module represented in the expression is 0, except for one section which must have a coefficient of +1 (positive relocation) and is immobilely associated with that section. An expression is relocatable if the coefficient for every section within the current module represented in the expression is 0, except for one section which must have a coefficient of +1 (positive relocation) and is relocatably associated with that section. An expression error occurs if a coefficient does not equal 0 or +1, or if more than one coefficient is nonzero.

An expression is external if the expression contains one external term and if the coefficients of all sections are 0. An expression error occurs if more than one external term is present. All external terms defined with the EXT pseudo have a value of 0 associated with them.

Examples:

```
|Location|Result    |Operand        |Comment
|1       |10        |20             |35
|        |          |               |
|        |IDENT     |TEST           |
|        |EXT       |EXT1           |
|SNAME1  |SECTION   |               |
|SYM1    |BSS       |4              |
|SYM2    |=         |W.*            |
|        |BSS       |5              |
|        |SYM4      |EXT1+SYM1      |; Illegal; it is not permitted
|        |          |               |; to have an external term and
|        |          |               |; relocatable terms with
|        |          |               |; coefficients of 1 in the same
|        |          |               |; expression.
|        |SYM5      |EXT1+SYM1-SYM2 |; Legal; the coefficients for
|        |          |               |; SYM1 (+1) and SYM2 (-1) cancel
|        |          |               |; each other and produce a
|        |          |               |; coefficient of 0 for the
|        |          |               |; expression.  The value of the
|        |          |               |; expression EXT1+SYM1-SYM2 is
|        |          |               |; 4 (0+0-4).
|        |END       |               |
```

See section 3, The CAL Program, for a description of sections.

4.9.2 PARCEL-ADDRESS, WORD-ADDRESS, OR VALUE ATTRIBUTES

Every expression assumes one of the following attributes:

- Parcel-address
- Word-address
- Value

An expression has parcel-address attribute if at least one term has a parcel-address attribute and all other terms have value or parcel-address attributes.

An expression has word-address attribute if at least one term has a word-address attribute and all other terms have value or word-address attributes.

All other expressions have value attributes. A warning message is issued if an expression has terms with both parcel-address and word-address attributes.

4.9.3 TRUNCATING EXPRESSION VALUES

An expression value is truncated to the field size of the expression destination.

Example:

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|SYM1    |BSS       |4            |;
|SYM2    |=         |-1           |; 64 bits
|        |VWD       |5/-1         |; 5 bits
|        |VWD       |3/5          |; 3-bit destination field,
|        |          |             |; value of 5
|        |VWD       |2/5          |; 2-bit destination field,
|        |          |             |; value of 5; truncation message
|        |          |             |; issued.
|        |VWD       |3/exp        |; 3-bit destination field,
|        |          |             |; the range of values is as
|        |          |             |; follows -4 ≤ exp ≥ 7
```

A warning message is issued if the left-most bits lost in truncation are not all zeros or all ones with the left-most remaining bit also one (that is, a negative quantity).

Truncation occurs in the statement VWD 5/-1 (diagram 4-13), but an error message is not generated because the part that was truncated included all ones and the left-most bit of the 5-bit field is also a one. The result of the VWD is stored in 5 bits as shown in diagram 4-14.

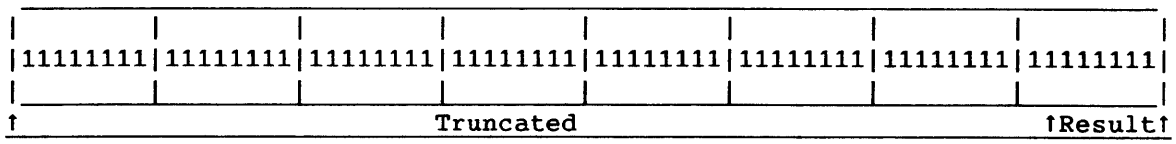

Diagram 4-13. 64-bit Representation of -1

Diagram 4-14. Truncated Value of -1 Stored in a 5-bit Field

Truncation occurs in the statement VWD 3/5. An error message is not generated, because the truncated part was all zeros. The result is stored as shown in diagram 4-15 and then truncated and stored as shown in diagram 4-16.

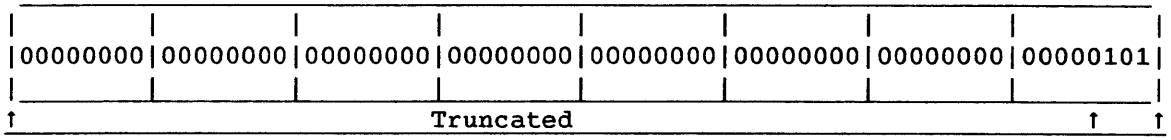

Diagram 4-15. 64-bit Representation of 5

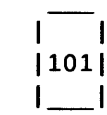

Diagram 4-16. Truncated Value of 5 Stored in a 3-bit Field

Truncation occurs in the statement VWD 2/5. CAL generates a warning message, because a combination of ones and zeros is truncated. The result is stored as shown in diagram 4-17 and then truncated and stored as shown in diagram 4-18.

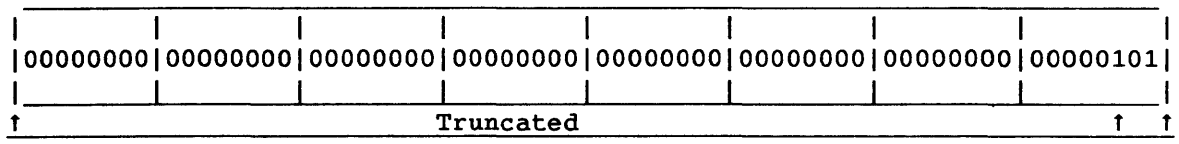

Diagram 4-17. 64-bit Representation of 5

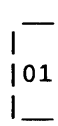

Diagram 4-18. Truncated Value of 5 Stored in a 2-bit Field

If the values generated by the statement VWD 3/*exp* are in the range from -4 through 7, a warning message is not generated.

Any message with a priority of error issued for an expression causes the expression to have a relative attribute of absolute, an address attribute of value, and a value of 0.

Examples of expressions:

Expression	Description
ALPHA	An expression consisting of a single term.
*W+BETA	Two terms; *W and BETA.
GAMMA/4+DELTA*5	Two terms; each consisting of two elements.
MU-NU*2+*	Three terms; the first consisting only of MU, the second consisting of NU*2, and the third consisting only of the special element *.
O'100+=O'100	Two terms; a constant and the address of a literal.

In the following examples, P and Q are immobile symbols in the same section, R and S are relocatable symbols in the same section, COM is relocatable in a common section, X and Y are external, and A and B are absolute. The location counter is currently in the section containing R and S.

The following expressions are absolute.

```
A+B
'A'R-1
2*R-S-*             Relocation of terms all cancel.
1/2*R  Equivalent to 0*R.
A*(R-S)             Error; parentheses not allowed.
```

The following expressions are immobile.

```
P+B
Q+3
COM+P-Q          P and Q cancel.
X+P              Error; external and immobile.
R+P              Error; relocatable and immobile.
P+Q              Error; immobile coefficient of 2.
Q/16*16          Error; division of immobile element is illegal.
```

The following expressions are relocatable.

```
*
W.*+B
R+2
COM+R-S          R and S cancel.
3**-R-S          3** cancels -R and -S.
=A'LITERAL'      Relocatable.
X+R              Error; external and relocatable.
R+S              Error; relocation coefficient of 2.
Q+S              Error; immobile and relocatable.
R/16*16          Error; division of relocatable element is illegal.
```

The following expressions are external.

```
X+2
Y-100
X+R-*            R, -* cancel relocation.
X+2**-R-S        Relocatable terms 2**, -R, -S cancel each other.
-X+2             Error; external cannot be negated.
X+Y              Error; more than one external.
X/Z              Error; division of an external element is illegal.
```

5. PSEUDO INSTRUCTIONS

Cray Assembly Language (CAL) includes a set of instructions known as pseudo instructions that direct the assembler in its task of interpreting the source statements and generating an object program.

Each program module begins with an IDENT pseudo instruction and ends with an END pseudo instruction. Symbol, micro, macro, and opdef definitions occurring within the program module are cleared before assembling the next program module.

A symbol, micro, macro, or opdef can be defined before the first IDENT pseudo instruction or between an END and a subsequent IDENT pseudo instruction. Such a definition is global and can be referenced in any subsequent program module. (Refer to Global Definitions, subsection 3.1.2.)

Redefinable micros and symbols can only be defined locally. Redefinable micros and symbols appearing before the first IDENT or between an END and subsequent IDENT pseudo instruction are cleared after assembling the next program module.

Symbolic machine instructions and the pseudo instructions listed below must appear within a program module. They are allowed outside of an IDENT to END sequence only within opdef or macro definitions.

ALIGN	BSS	CON	LOC	START
BITP	BSSZ	DATA	ORG	VWD
BITW	COMMENT	ENTRY	QUAL	
BLOCK	COMMON	EXT	SECTION	

The LOCAL pseudo instruction must occur after a macro or opdef prototype statement or DUP or ECHO pseudo instructions, except for intervening comment statements. All other pseudo instructions, macro definitions, and opdef definitions can appear anywhere.

Pseudo instructions are classified and described according to their applications, as follows:

Class	Pseudo Instructions
Program control	IDENT, END, COMMENT
Loader linkage	ENTRY, EXT, START
Mode control	BASE, QUAL, EDIT, FORMAT
Section control	SECTION, BLOCK, COMMON, ORG, LOC, BITW, BITP, BSS, ALIGN
Message control	ERROR, ERRIF, MLEVEL, DMSG

Class	Pseudo Instructions
Listing control	LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, ENDTEXT
Symbol definition	=, SET, MICSIZE
Data definition	CON, BSSZ, DATA, VWD
Conditional assembly	IFA, IFC, IFE, IFM, ENDIF, SKIP, ELSE
Micro definition	CMICRO, MICRO, OCTMIC, DECMIC
File control	INCLUDE
Defined sequences	MACRO, OPDEF, DUP, ECHO, ENDM, ENDDUP, STOPDUP, LOCAL, OPSYN, EXITM, NEXTDUP

NOTE

Pseudo instructions can be specified in uppercase or lowercase, but never in mixed case.

The syntax for pseudos is not presented in strict Backus-Naur Form (BNF). In some cases, the BNF has been condensed to eliminate unnecessary redundancy in the documentation.

Throughout this section, pseudos with ignored fields (location or operand) are defined as follows:

Location	Result	Operand
ignored	*pseudox*	

pseudox Pseudo instruction with a blank location field

ignored The location field of this statement is ignored by the assembler. A message with a priority of CAUTION is issued if the field is not empty and all of the characters in the field are skipped until a blank character is encountered. The first nonblank character following the blank character is assumed to be the beginning of the result field.

Location	Result	Operand
	pseudoy	ignored

pseudoy Pseudo instruction with a blank operand field

ignored The operand field of this statement is ignored by the assembler. A message with a priority of CAUTION is issued if the field is not empty and all of the characters in the field are skipped until a blank character is encountered. The first nonblank character following the blank character is assumed to be the beginning of the comment field.

5.1 PROGRAM CONTROL

The pseudo instructions described in this subsection define the limits of a program module.

- IDENT Marks the beginning of a program module
- END Marks the end of a program module
- COMMENT Enters comment, generally a copyright, into the generated binary load module.

5.1.1 IDENT - IDENTIFY PROGRAM MODULE

The IDENT pseudo instruction identifies a program module and marks its beginning. The name of the module appears in the heading of the listing produced by CAL (if the title pseudo has not been used) and in the generated binary load module.

The IDENT pseudo must be specified in the global part of a CAL program. If the IDENT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the IDENT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	IDENT	*name*
ignored	ident	*name*

name Name of the program module. *name* must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2, Names.

Example:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	EXAMPLE	

5.1.2 END - END PROGRAM MODULE

The END pseudo instruction terminates a program segment (module initiated with an IDENT pseudo) under the following conditions:

- If the assembler is not in definition mode
- If the assembler is not in skipping mode
- If the END pseudo does not occur within an expansion

If the END pseudo is found within a definition, a skip sequence, or an expansion, a message is issued indicating that the pseudo is not allowed within these modes and the statement is treated as follows:

- Defined if in definition mode
- Skipped if in skipping mode
- Do-nothing instruction if in an expansion

The END pseudo instruction can be specified from within a program module only. If the END pseudo instruction validly terminates a program module, it causes the assembler to take the following actions:

- Generate a cross reference for symbols if the cross reference list option is enabled, and the listing is enabled
- Clear and reset the format option
- Clear and reset the edit option
- Clear and reset the message priority
- Clear and reset all list control options
- Clear and reset the default numeric base
- Discard all qualified, redefinable, nonglobal, and %% symbols
- Discard all qualifiers

- Discard all redefinable and nonglobal micros
- Discard all local macros, opdefs, and local pseudos (defined with an OPSYN pseudo)
- Discard all sections

Format:

Location	Result	Operand
ignored	END	ignored
ignored	end	ignored

5.1.3 COMMENT - ENTER COMMENT INTO GENERATED BINARY LOAD MODULE

The COMMENT† pseudo instruction defines a character string of up to 256 characters to be entered as an informational comment in the generated binary load module.

If the operand field is empty, the comment field is cleared and no comment is generated. If a comment is specified more than once, the last specification is used. If a comment is specified more than once and the current comment is different from the previous comment, a message with a priority of caution is issued.

If a subprogram contains more than one COMMENT pseudo, the character string from the last COMMENT pseudo is inserted into the binary load module.

The COMMENT pseudo instruction must be specified from within a program module. If the COMMENT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the COMMENT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	COMMENT	[*del-char*[*string-of-ASCII*]*del-char*]
ignored	comment	[*del-char*[*string-of-ASCII*]*del-char*]

† CRAY-1 and CRAY X-MP Computer Systems only

string-of-ASCII
An optional ASCII character string of any length.

del-char Delimiter character; must be a single matching character on both ends of the ASCII character string. A character string can be delimited by a character other than an apostrophe. Any ASCII character other than a space can be used. Two consecutive occurrences of the delimiting character indicate a single such character is to be included in the character string.

Example:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |IDENT     |CAL              |
|        |COMMENT   |'COPYRIGHT CRAY RESEARCH, INC. 1985'
|        |COMMENT   |-CRAY X-MP Computer System-
|        |COMMENT   |@ABCDEF@@FEDCBA@
|        |END       |                 |
```

5.2 LOADER LINKAGE

The pseudo instructions described in this subsection provide for loading multiple object program modules, linking them into a single executable program (ENTRY and EXT), and specifying the main program entry (START).

- ENTRY Specifies symbols, defined as addresses or values, so they can be used by other program modules linked by a loader
- EXT Specifies linkage to addresses or values defined as entry symbols in other program modules
- START Specifies symbolic address where execution begins

5.2.1 ENTRY - SPECIFY ENTRY SYMBOLS

The ENTRY pseudo instruction specifies symbolic addresses or values that can be referred to by other program modules linked by the loader. Each entry symbol must be an absolute, immobile, or relocatable symbol defined within the program module.

The ENTRY pseudo instruction is restricted to sections that allow instructions, data, or instructions and data. If the ENTRY pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ENTRY pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	ENTRY	[*symbol*]{","[*symbol*]}
ignored	entry	[*symbol*]{","[*symbol*]}

symbol Name of zero, one, or more symbols; each of the names must be defined as an unqualified symbol within the same program module. The corresponding symbol must not be redefinable, external, or relocatable relative to either a stack or a task common section.

symbol must meet the requirements for symbols as described in the BNF. For a description of symbols, see subsection 4.3.

Example:

Location	Result	Operand	Comment
1	10	20	35
	ENTRY	EPTNME,TREG	
	.		
	.		
	.		
EPTNME	=	*	
TREG	=	O'17	

5.2.2 EXT - SPECIFY EXTERNAL SYMBOLS

The EXT pseudo instruction specifies linkage to symbols that are defined as entry symbols in other program modules. They can be referred to from within the program module but must not be defined as unqualified symbols elsewhere within the program module. Symbols specified on the EXT instruction are defined as unqualified symbols having relative attributes of external and the specified address attribute.

The EXT pseudo instruction can be specified anywhere within a program module. If the EXT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the EXT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	EXT	[*symbol*{":"[*attribute*]}]{","[*symbol*{":"[*attribute*]}]}
ignored	ext	[*symbol*{":"[*attribute*]}]{","[*symbol*{":"[*attribute*]}]}

symbol Name of zero, one, or more external symbols; each must be an unqualified symbol having a relative attribute of external and the corresponding address attribute.

symbol must meet the requirements for symbols as described in the BNF. For a description of symbols, see subsection 4.3.

attribute Specify one of the following attribute types: *address-attribute* or *linkage-attribute*

address-attribute is the address attribute to be assigned to the external symbol; can be one of the following:

V or v Value (default)
P or p Parcel
W or w Word

linkage-attribute is the linkage attribute to be assigned to the external symbol. Linkage attributes can be specified in uppercase, lowercase, or mixed case and can be one of the following:

HARD (default)
SOFT

If the *linkage-attribute* is not specified on the EXT pseudo, the default is HARD. All hard external references are resolved at load time.

A soft reference for a particular external name is resolved at load time if and only if at least one other module has referenced that same external name as a hard reference.

A user conditionally references a soft external name at execution time. If a soft external name has not been included at load time and is referenced at execution time, an appropriate message is issued.

If the operating system for which the assembler is generating code does not support soft externals, a caution level message is issued and soft externals are treated as hard externals.

NOTE

SOFT saves memory and time by excluding software packages, such as graphic routines, debugging routines, and so on, that may be available on your system but are not required by your program. HARD is, however, recommended for most users.

Example:

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|        |IDENT     |A         |
|        |.         |          |
|        |.         |          |
|        |.         |          |
|        |ENTRY     |VALUE     |
|VALUE   |=         |2.0       |
|        |.         |          |
|        |.         |          |
|        |.         |          |
|        |END       |          |
|        |IDENT     |B         |
|        |EXT       |VALUE     |
|        |CON       |VALUE     |; The 64-bit external value 2.0
|        |.         |          |; is stored here by the loader.
|        |.         |          |
|        |.         |          |
|        |END       |          |
```

5.2.3 START - SPECIFY PROGRAM ENTRY

The START pseudo instruction specifies the main program entry. The program uses the START pseudo to specify the symbolic address where execution begins following the loading of the program. The named symbol can optionally be an entry symbol specified in an ENTRY pseudo instruction.

The START pseudo instruction must be specified from within a program module. If the START pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the START pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	START	*symbol*
ignored	start	*symbol*

symbol Name of a symbol; must be defined as an unqualified symbol within the same program module. *symbol* must not be redefinable, must have a relative attribute of relocatable, and cannot be relocatable relative to any section other than a section that allows instructions or a section that allows instructions and data. The START pseudo cannot be specified in a section with a type of data only.

symbol must meet the requirements for symbols as described in the BNF. For a description of symbols, see subsection 4.3.

Example:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	EXAMPLE	
	START	HERE	
HERE	=	*	
	.		
	.		
	.		
	END		

5.3 MODE CONTROL

Mode control pseudo instructions define the characteristics of an assembly. The BASE pseudo determines whether notation for numeric data is assumed to be octal or decimal. The QUAL pseudo instruction permits symbols to be defined as qualified or unqualified. The EDIT pseudo instruction controls editing of assembler statements. The FORMAT pseudo instruction controls the format that is used for interpreting assembly source statements.

- BASE Specifies data as being octal, decimal, or a mixture of both
- QUAL Designates a sequence of code where symbols may be defined with a qualifier, such as a common routine with its own labels
- EDIT Turns editing on or off
- FORMAT Changes the format to old or new

5.3.1 BASE - DECLARE BASE FOR NUMERIC DATA

The BASE pseudo instruction allows specification of the base of numeric data as being octal, decimal, or mixed when the base is not explicitly specified by an O', D' or H' prefix. The default is decimal.

The BASE pseudo instruction can be specified anywhere in a program segment. If the BASE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the BASE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	BASE	*option*
ignored	BASE	*
ignored	base	*option*
ignored	base	*

option Numeric base used for the integer. *option* is a required single character as follows:

O or o Octal
D or d Decimal (default mode)
M or m Mixed; numeric data is assumed to be octal, except for numeric data used for the following, which is assumed to be decimal:

- Statement counts in DUP and conditional statements
- Line count in the SPACE pseudo instruction
- Bit position or count in the BITW, BITP, or VWD pseudo instructions
- Character counts as in CMICRO, MICRO, OCTMIC, and DECMIC pseudo instructions
- Character count in data items (see data in section 4, Cray Assembly Language)

* Reverts to the prefix that was in effect prior to the specification of the current prefix within the current program segment. Each occurrence of a BASE pseudo instruction other than BASE * can modify the current prefix. Each BASE * releases the most current prefix and reactivates the prefix that preceded the current prefix. If all BASE pseudos specified have been released, a message with a priority of CAUTION is issued, and the default mode (decimal) is used.

Example:

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|        |BASE      |O         |; Change base from default to
|        |          |          |; octal
|        |VWD       |50/12     |; Field size and constant
|        |.         |.         |; value both octal
|        |.         |.         |
|        |.         |.         |
|        |BASE      |D         |; Change base from octal to
|        |          |          |; decimal
|        |VWD       |49/19     |; Field size and constant
|        |.         |.         |; value both decimal
|        |.         |.         |
|        |.         |.         |
```

Example (continued):

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |BASE      |M           |; Change from decimal to mixed
|        |          |            |; base
|        |VWD       |39/12       |; Field size decimal; constant
|        |.         |.           |; value octal.
|        |.         |.           |
|        |.         |.           |
|        |BASE      |*           |; Resume decimal base
|        |BASE      |*           |; Resume octal base
|        |BASE      |*           |; Stack empty; resume decimal
|        |          |            |; base.
```

5.3.2 QUAL - QUALIFY SYMBOLS

A QUAL pseudo instruction begins or ends a code sequence in which all symbols defined either are qualified by a qualifier specified by the QUAL or are unqualified. Until the first use of a QUAL pseudo instruction, symbols are defined as unqualified for each program segment. Global symbols cannot be qualified. Thus, QUAL pseudo instructions must not occur before an IDENT pseudo instruction.

A qualifier applies to symbols only. Names used for sections, conditional sequences, duplicated sequences, macros, micros, externals, formal parameters, and so on, are not affected.

The QUAL pseudo instruction must be specified from within a program module. If the QUAL pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the QUAL pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

At the end of each program segment all qualified symbols are discarded.

Format:

```
|Location|Result    |Operand
|        |          |
|ignored |QUAL      |[name]
|ignored |QUAL      |*
|        |          |
|ignored |qual      |[symbol]
|ignored |qual      |*
```

name Optional name qualifier. Indicates whether symbols are to be qualified or unqualified and, if qualified, indicates the qualifier to be used. *name* must meet the requirements for names as described in the BNF. For a description of names and qualifiers, see section 4, Cray Assembly Language.

name causes all symbols defined until the next QUAL pseudo instruction to be qualified. A qualified symbol can be referenced with or without the qualifier that is currently active. If the symbol is referenced while some other qualifier is active, the reference must be in the following form:

/qualifier/symbol

When a symbol is referenced without a *qualifier*, CAL attempts to find it in the currently active *qualifier*. If the qualified symbol is not defined within the current qualifier, CAL attempts to find it in the list of unqualified symbols. The symbol is undefined if both of these searches fail.

An unqualified symbol can explicitly be referenced using the following form:

//symbol

If the operand field of the QUAL is empty, symbols are unqualified until the next occurrence of a QUAL pseudo instruction. An unqualified symbol can be referenced without qualification from any place in the program module, or in the case of global symbols, from any program segment assembled after the symbol definition.

* An * resumes use of the qualifier in effect before the most recent qualification within the current program segment. Each occurrence of a QUAL other than a QUAL * causes the initiation of a new qualifier. Each QUAL * removes the current qualifier and causes the most recent prior qualification to be activated. If the QUAL * statement is encountered and all specified qualifiers have been released, a message with a priority of CAUTION is issued and succeeding symbols are defined as being unqualified.

Example:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |.         |                 |; Assembler default for symbols
|        |.         |                 |; is unqualified.
|        |.         |                 |
|ABC     |=         |1                |; ABC is unqualified.
|        |QUAL      |QNAME1           |; Symbol qualifier QNAME1.
|ABC     |=         |2                |; ABC is qualified by QNAME1.
|        |J         |XYZ              |
|XYZ     |S1        |A2               |; XYZ is qualified by QNAME1
|        |.         |                 |
|        |.         |                 |
|        |.         |                 |
|        |QUAL      |QNAME2           |; Symbol qualifier QNAME2.
|ABC     |=         |3                |
|        |J         |/QNAME1/XYZ      |
|        |.         |                 |
|        |.         |                 |
|        |.         |                 |
|        |QUAL      |*                |; Resume the use of symbols
|        |          |                 |; qualified with qualifier
|        |          |                 |; QNAME1.
|        |.         |                 |
|        |.         |                 |
|        |.         |                 |
|        |QUAL      |*                |; Resume the use of unqualified
|        |          |                 |; symbols.
|A       |IFA       |DEF,ABC          |; Test whether ABC is defined
|B       |IFA       |DEF,/QNAME1/ABC  |; Test whether ABC is defined
|        |          |                 |; within qualifier QNAME1.
|C       |IFA       |DEF,/QNAME2/ABC  |; Test for /QNAME2/ABC being
|        |          |                 |; defined within qualifier
|        |          |                 |; QNAME2
|        |.         |                 |
|        |.         |                 |
|        |.         |                 |
```

5.3.3 EDIT - CHANGE STATEMENT EDITING STATUS

The EDIT pseudo allows you to turn editing off and on within a program segment. Appending (^ - new format) and continuation (, - old format) are not effected by the EDIT pseudo. The current editing status is reset at the beginning of each segment to the editing option specified on the CAL invocation statement. See section 3, The CAL Program, for a description of statement editing.

The EDIT pseudo can be specified anywhere within a program segment. If the EDIT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the EDIT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	EDIT	[*option*]
ignored	EDIT	*
ignored	edit	[*option*]
ignored	edit	*

option Turns editing on and off. *option* can be specified in uppercase, lowercase, or mixed case and can be one of the following:

ON	Editing is enabled.
OFF	Editing is disabled.
No entry	Editing is enabled.

* An * resumes use of the edit option in effect before the most recent edit option within the current program segment. Each occurrence of an EDIT other than an EDIT * causes the initiation of a new edit option. Each EDIT * removes the current edit option and reactivates the edit option that preceded the current edit option. If the EDIT * statement is encountered and all specified edit options have been released, a message with a priority of CAUTION is issued and the default, ON, is used.

5.3.4 FORMAT - CHANGE STATEMENT FORMAT

CAL Version 2 supports both the CAL Version 1 statement format and a new statement format. The FORMAT pseudo allows you to switch between statement formats within a program segment. The current statement format is reset at the beginning of each section to the format option specified on the CAL invocation statement. For a description of the recommended formatting conventions for the new format, see section 3, The CAL Program.

The FORMAT pseudo can be specified anywhere within a program segment. If the FORMAT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the FORMAT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	FORMAT	[*option*]
ignored	FORMAT	*
ignored	format	[*option*]
ignored	format	*

option Specifies old or new format. *option* can be specified in uppercase, lowercase, or mixed case and can be one of the following:

OLD	Old format
NEW	New format
No entry	Statement format reverts to the format specified on the CAL invocation statement. See section 2, Operating Systems, for a description of the options available with the CAL invocation statement.

* An * resumes use of the format option in effect before the most recent format option within the current program segment. Each occurrence of a FORMAT other than a FORMAT * causes the initiation of a new format option. Each FORMAT * removes the current format option and reactivates the format that preceded the current format. If the FORMAT * statement is encountered and all specified format options have been released, a message with a priority of CAUTION is issued and the default is used.

5.4 SECTION CONTROL

Section control pseudo instructions control the use of sections and counters in a CAL program.

- SECTION Defines specific program sections and replaces the BLOCK and COMMON pseudos. SECTION is recommended over the BLOCK and COMMON pseudo instructions because it has all of the capablilities of BLOCK and COMMON plus many other capabilities not available to BLOCK and COMMON.

- BLOCK† Defines local sections

- COMMON† Defines common sections that can be referenced by another program module

- STACK Increments the size of the stack

- ORG Resets location and orgin counters

- LOC Resets location counter

- BSS Reserves memory

- BITW Sets the current bit position relative to the current word

- BITP Sets the current bit position relative to the current parcel

- ALIGN† Aligns code on an instruction buffer boundary

5.4.1 SECTION - SECTION ASSIGNMENT

The SECTION pseudo instruction establishes or resumes a section of code. The section may be common or local, depending on the options found in the operand field. Each section has its own location, origin, and bit position counters.

The SECTION pseudo instruction must be specified from within a program module. If the SECTION pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SECTION pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

† Available on the CRAY X-MP and CRAY-1 Computer Systems only

Format:

Location	Result	Operand
[*name*]	SECTION	[*type*]["," [*location*]]
[*name*]	SECTION	[*location*]["," [*type*]]
[*name*]	SECTION	*
[*name*]	section	[*type*]["," [*location*]]
[*name*]	section	[*location*]["," [*type*]]
[*name*]	section	*

name Optional name of the section. *name* must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

type Type of section; can be specified in upper, lower, or mixed case. *type* can be one of the following:

MIXED - Defines a section that permits both instructions and data. MIXED is the default type for the main section initiated by the IDENT pseudo. If *type* is not specified, MIXED is the default. A MIXED section is treated as a local section by the loader. For a description of local sections, see subsection 3.6.1.

CODE - Restricts a section to instructions only; data is not permitted. A CODE section is treated as a local section by the loader. For a description of local sections, see subsection 3.6.1.

DATA - Restricts a section to to data only (CON, DATA, BSSZ, and so on); instructions are not permitted. The DATA section is treated as a local section by the loader. For a description of local sections, see subsection 3.6.1.

STACK - Sets up a stack frame (designated memory area). Neither data nor instructions are allowed. All symbols that are defined using the location or origin counter and are relative to a section that has a type of STACK are assigned a relative attribute of immobile. These symbols may be used as offsets into the STACK section itself. These sections are treated like other section types except relocation does not occur after assembly. Since relocation does not occur, sections with a type of stack are not passed to the loader.

Sections with a type of STACK conveniently indicate that symbols are relative to an execution-time stack frame and that their values correspond to an absolute location within the stack frame relative to the base of the stack frame. Symbols with stack attributes are indicated as such in the debug tables produced by CAL. For a description of local sections, see subsection 3.6.1.

NOTE

Accessing data from a stack section is not as straight forward as accessing data directly from memory. For more information about stacks, see the Macros and Opdefs section of the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual, publication SR-2013, or the CRAY X-MP and CRAY-1 Computer Systems Macros and Opdefs Reference Manual, CRI publication SR-0012.

COMMON - Defines a common section that can be referenced by another program module. Instructions are not allowed.

Data cannot be defined in a COMMON section without a name (no name in location field); only storage reservation can be defined in an unnamed COMMON section. The location field that names a COMMON section cannot match the location field name of a previously defined section with a type of COMMON, DYNAMIC, or TASKCOM. If duplicate location field names are specified, a message with a priority of error is issued.

For a description of unnamed (blank) COMMON, see subsection 3.6.2, Common Sections.

DYNAMIC - Allocates an expandable common section at load time. DYNAMIC is a common section. Neither instructions nor data are permitted within a DYNAMIC section; only storage reservation can be defined in an unnamed DYNAMIC section. The location field that names a DYNAMIC section cannot match the location field name of a previously defined section with a type of COMMON, DYNAMIC, or TASKCOM. If duplicate location field names are specified, a message with a priority of error is issued.

For a description of blank DYNAMIC, see subsection 3.6.2, Common Sections.

TASKCOM† - Defines a task common section. Neither instructions nor data are allowed at assembly time. At execution time, TASKCOM is set up and can be referenced by all subroutines local to a task. Data can also be inserted at execution time into a TASKCOM section by any subroutine that is executed within a single task.

When a section is defined with a type of TASKCOM, CAL creates a symbol that is assigned the name in the location field of the SECTION pseudo defining the section. This symbol is not redefinable, has a value of zero, an address attribute of word, and a relative attribute that is relocatable relative to the section. This symbol is relocated by the loader and is used as an offset into an execution time task common table. The word at which it points within this table will contain the address of the base of the task common section in memory.

All symbols that are defined using the location or origin counter within a task common section are assigned a relative attribute of immobile. These symbols are treated like other symbols except relocation does not occur after assembly. These symbols may be used as an offset into the task common section itself.

Sections with a type of TASKCOM indicate that their symbols are relative to an execution-time task common section, and their values correspond to an absolute location within the task common section relative to the beginning of the task common section. These values are indicated as such in the debug tables produced by CAL. For a description of local sections, see subsection 3.6.1.

TASKCOM must always be named. The location field that names a TASKCOM section cannot match the location field name of a previously defined section with a type of COMMON, DYNAMIC, or TASKCOM. If duplicate location field names are specified, a message with a priority of error is issued.

NOTE

Accessing data from a task common section is not as straight forward as accessing data directly from memory. For more information about task common, see publication SN-0222, CRAY X-MP Multitasking Programmer's Manual.

---

† Available on the CRAY X-MP and CRAY-1 Computer Systems only

location The kind of memory to which the section is assigned, can be upper, lower, or mixed case, and must be one of the following:

CM	Central or Common Memory. This is the default.
EM†	Extended Memory
LM††	Local Memory

* The *name, type,* and *location* of the section in control reverts to the *name, type,* and *location* of the section in effect before the current section was specified within the current program module. Each occurrence of a SECTION pseudo instruction other than SECTION * causes a section with the *name, type,* and *location* specified to be allocated. Each SECTION * releases the currently active section and reactivates the section that preceded the current section. If all specified sections have been released when a SECTION * is encountered, CAL issues a message with a priority of CAUTION and uses the main section.

When *type* and/or *location* are not specified, MIXED and Common Memory are used by default.

If *type, location,* or *type* and *location* are not specified, the defaults are MIXED for *type* and CM for *location*. Since a module within a program segment is initialized without a name, with a *type* of MIXED, and with a *location* of CM, a SECTION pseudo instruction used without the specified location and operand fields forces this initial section entry to become the current working section.

If the section name and attributes have been previously defined, the SECTION pseudo makes the previously defined section entry the current working section. If the section name and attributes have not been defined, the SECTION pseudo attempts to create a new section with the name and attributes. The following restrictions apply when a new section is created:

- A type of TASKCOM must always have a location field name.
- If a section with a type of COMMON, DYNAMIC, or TASKCOM is being created for the first time, it must never have a name that matches a section that was created previously with a type of COMMON, DYNAMIC, or TASKCOM.

---

† Available on CRAY X-MP Computer Systems only
†† Available on CRAY-2 Computer Systems only

Example:

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|        |ident     |exsect        |; The Main section has by
|        |          |              |; default a type of mixed and a
|        |          |              |; location of Common Memory.
|        |.         |              |
|        |.         |              |
|        |.         |              |
|        |con       |1             |; Data and instructions are
|        |          |              |; permitted in the main section.
|        |s1        |1             |; Data and instructions are
|        |          |              |; permitted in the main section.
|        |.         |              |
|        |.         |              |
|        |.         |              |
|dsect   |section   |data          |; This section is defined with a
|        |          |              |; name of dsect, type of data,
|        |          |              |; and a location of Common
|        |          |              |; Memory.
|        |.         |.             |
|        |.         |.             |
|        |.         |.             |
|        |con       |3             |; Data is permitted in dsect.
|        |bssz      |2             |; Data is permitted in dsect.
|        |.         |.             |
|        |.         |.             |
|        |.         |.             |
|        |s2        |s3            |; CAL generates a message with
|        |          |              |; a priority of error, because
|        |          |              |; instructions are not permitted
|        |          |              |; in a section with a type of
|        |          |              |; data.
|        |.         |.             |
|        |.         |.             |
|        |.         |.             |
|csect   |section   |common        |; This section is defined with
|        |          |              |; a name of csect, a type of
|        |          |              |; common, and by default a
|        |          |              |; location of Common Memory.
|        |.         |.             |
|        |.         |.             |
|        |.         |.             |
|        |data      |'12345678'    |; Data is permitted in a named
|        |          |              |; common section.
```

Example (continued):

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|        |s2        |a1           |; CAL generates a message with
|        |          |             |; a priority of error, because
|        |          |             |; instructions are not permitted
|        |          |             |; in a common section.
|        |.         |.            |
|        |.         |.            |
|        |.         |.            |
|        |section   |             |; This section is unnamed and
|        |          |             |; is assigned by default a type
|        |          |             |; of mixed and a location of
|        |          |             |; Common Memory.  When a
|        |          |             |; section is specified without
|        |          |             |; a name, a type, and a
|        |          |             |; location, the main section
|        |          |             |; becomes the current section.
|        |.         |.            |
|        |.         |.            |
|        |.         |.            |
|        |section   |*            |; The current section reverts
|        |          |             |; to the previous section in
|        |          |             |; the stack buffer; csect.
|        |section   |*            |; The current section reverts
|        |          |-            |; to the previous section in
|        |          |             |; the stack buffer; dsect.
|        |.         |.            |
|        |.         |.            |
|        |.         |.            |
|        |con       |2            |; A memory location with a value
|        |          |             |; of 2 is inserted into dsect.
|        |.         |.            |
|        |.         |.            |
|        |.         |.            |
|        |section   |*            |; The current section reverts
|        |          |             |; to the main section.
|        |.         |.            |
|        |.         |.            |
|        |.         |.            |
|csect   |section   |common,cm    |; The current section reverts
|        |          |             |; to the section defined
|        |          |             |; previously as csect.  When a
|        |          |             |; section is specified with the
|        |          |             |; name, type, and location of a
|        |          |             |; previously defined section,
|        |          |             |; the previously defined section
|        |          |             |; becomes the current section.
```

Example (continued):

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|        |.         |.         |
|        |.         |.         |
|        |.         |.         |
|        |section   |*         |; The current section reverts
|        |          |          |; to the main section.
|        |.         |.         |
|        |.         |.         |
|        |.         |.         |
|dsect   |section   |code      |; CAL considers this section
|        |          |          |; specification unique and
|        |          |          |; different from the previously
|        |          |          |; defined section named dsect.
|        |          |          |; Sections with types of mixed,
|        |          |          |; code, data, and stack are
|        |          |          |; treated as local sections by
|        |          |          |; the loader.  Local sections
|        |          |          |; that are specified with the
|        |          |          |; same name are, therefore,
|        |          |          |; considered unique if they are
|        |          |          |; specified with different
|        |          |          |; types.
|        |.         |.         |
|        |.         |.         |
|        |.         |.         |
|        |s1        |s2        |; Instructions are permitted in
|        |          |          |; dsect.
|        |.         |.         |
|        |.         |.         |
|        |.         |.         |
|        |con       |2         |; CAL generates a message with
|        |          |          |; a priority of error because
|        |          |          |; data is not permitted in a
|        |          |          |; section with a type of code.
|        |.         |.         |
|        |.         |.         |
|        |.         |.         |
|        |section   |*         |; The current section reverts
|        |          |          |; to the main section.
|        |.         |.         |
|        |.         |.         |
|        |.         |.         |
```

Example (continued):

```
|Location|Result    |Operand        |Comment
|1       |10        |20             |35
|        |          |               |
|csect   |section   |dynamic        |; CAL generates a message with
|        |          |               |; a priority of error, because
|        |          |               |; sections with types of common,
|        |          |               |; dynamic, and taskcom are not
|        |          |               |; treated as local sections by
|        |          |               |; the loader.  Specifying a
|        |          |               |; section with a previously
|        |          |               |; defined name is illegal when
|        |          |               |; the accompanying type does
|        |          |               |; not define a local section.
|        |end       |               |
```

5.4.2 BLOCK† - LOCAL SECTION ASSIGNMENT

The BLOCK pseudo instruction establishes or resumes use of a local section of code within a program module. Each section has its own location, origin, and bit position counters.

This pseudo defines a mixed local section in which both code and/or data can be stored. The section is assigned to central, or common, memory. See the SECTION pseudo in this section for more information.

The BLOCK pseudo instruction must be specified from within a program module. If the BLOCK pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the BLOCK pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result    |Operand
|        |          |
|        |BLOCK     |[name]
|        |BLOCK     |*
|        |          |
|        |block     |[name]
|        |block     |*
```

† Available on CRAY X-MP and CRAY-1 Computer Systems only

name Optional block name; indicates which section is used for assembling code until the occurrence of the next BLOCK or COMMON pseudo instruction.

name must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

* The section in control reverts to the section in effect before the current section was specified within the current program module. Each occurrence of a BLOCK pseudo instruction other than BLOCK * causes a section to be allocated. Each BLOCK * releases the currently active section and reactivates the section that preceded the current section. If all specified sections have been released when a BLOCK * is encountered, CAL issues a message with a priority of CAUTION and uses the main section.

Example:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |          |            |
|        |.         |            |; Main section in use
|        |.         |            |
|        |.         |            |
|        |BLOCK     |A           |; Use section A
|        |.         |            |
|        |.         |            |
|        |.         |            |
|        |BLOCK     |            |; Use main section
|        |.         |            |
|        |.         |            |
|        |.         |            |
|        |BLOCK     |*           |; Return to use of section A
```

5.4.3 COMMON† - COMMON SECTION ASSIGNMENT

The COMMON pseudo instruction establishes a common section or resumes a previous section. Each section has its own location, origin, and bit position counters.

† Available on the CRAY X-MP and CRAY-1 Computer Systems only

This pseudo defines a common section that can be referenced by another program module. Instructions are not allowed. The section is assigned to central, or common, memory. See the SECTION pseudo in this section for more information.

Data cannot be defined in a COMMON section without a name (no name in location field); only storage reservation can be defined in an unnamed COMMON section. The location field that names a common section cannot match the location field name of a previously defined section with a type of COMMON, DYNAMIC, or TASKCOM. If duplicate location field names are specified, a message with a priority of error is issued.

For a description of unnamed (blank) COMMON, see section 3, The CAL Program.

The COMMON pseudo instruction must be specified from within a program module. If the COMMON pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the COMMON pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	COMMON	[*name*]
ignored	COMMON	*
ignored	common	[*name*]
ignored	common	*

name Optional name of the common section to be defined. *name* must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

Unlabeled common sections have specific restrictions. For a detailed description of blank COMMON sections, see section 3, The CAL Program.

* The section in control reverts to the section in effect before the current section was specified within the current program module. Each occurrence of a COMMON pseudo instruction other than COMMON * causes a section to be allocated. Each COMMON * releases the currently active section and reactivates the section that preceded the current section. If all specified sections have been released when a COMMON * is encountered, CAL issues a message with a priority of CAUTION and uses the main section.

Example:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |.         |                 |; Main section
|        |.         |                 |
|        |.         |                 |
|        |.         |                 |
|        |COMMON    |FIRST            |; Labeled common section FIRST
|        |.         |                 |
|        |.         |                 |
|        |.         |                 |
|        |COMMON    |                 |; Blank common
|        |.         |                 |
|        |.         |                 |
|        |.         |                 |
|        |COMMON    |*                |; Return to labeled common
|        |          |                 |; section FIRST
|        |.         |                 |
|        |.         |                 |
|        |.         |                 |
|        |COMMON    |*                |; Return to the main section
```

5.4.4 STACK - INCREMENT THE SIZE OF THE STACK

The STACK pseudo increases the size of the stack. Increments made by the STACK pseudo are cumulative. Each time the STACK pseudo is used within a module, the current stack size is incremented by the number of words specified by the expression in the operand field of the STACK pseudo.

The STACK pseudo is used in conjunction with sections that have a type of STACK. If either a STACK section or the STACK pseudo is specified within a module, the loader tables produced by the assembler indicate that the module uses one or more stacks. The stack size indicated in the loader tables is the combined sizes of all STACK sections, if any, added to the total value of all STACK pseudos, if any, specified within a module.

The STACK pseudo instruction must be specified from within a program module. If the STACK pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the STACK pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	STACK	[*expression*]
ignored	stack	[*expression*]

expression

Optional expression. If *expression* is specified, it must have an address attribute of word or value, a relative attribute of absolute, a positive value, and all symbols within it, if any, must have been previously defined.

If STACK is specified without *expression*, the stack is not incremented.

expression must meet the requirements for an expression as described in the BNF. For a description of expressions, see subsection 4.7.

5.4.5 ORG - SET * AND *O COUNTER

The ORG pseudo instruction resets the location and origin counters to the value specified. ORG resets the location and origin counters to the same value relative to the same section.

The ORG pseudo instruction forces a word boundary within the current section and also within the new section specified by the expression. These force word boundaries occur before the counter is reset. ORG can change the current working section in use without modifying the section stack.

The ORG pseudo instruction is restricted to sections that allow instructions, data, or instructions and data. If the ORG pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ORG pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	ORG	[*expression*]
ignored	org	[*expression*]

expression

An optional immobile or relocatable expression with positive relocation within the section currently in use. If the expression is blank, the word address of the next available word in the section is used. A force word boundary occurs before the expression is evaluated.

The expression must have a value or word-address attribute. If the expression has a value attribute, it is assumed to be a word address. If the expression exists, all symbols, if any, must be previously defined. If the current base is mixed, octal is used as the base.

The expression cannot have any of the following: an address attribute of parcel, a relative attribute of absolute or external, or a negative value.

expression must meet the requirements for an expression as described in the BNF. For a detailed description of expressions, see subsection 4.7.

Example:

Location	Result	Operand	Comment
1	10	20	35
	ORG	W.*+O'200	

5.4.6 BSS - BLOCK SAVE

The BSS pseudo instruction reserves a block of memory in a section. A force word boundary occurs and then the number of words specified by the operand field expression is reserved. Data is not generated by this pseudo instruction. The block of memory is reserved by increasing the location and origin counters.

The BSS pseudo instruction must be specified from within a program module. If the BSS pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the BSS pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*symbol*]	BSS	[*expression*]
[*symbol*]	bss	[*expression*]

symbol Optional symbol. Assigned the word address of the location counter after the force word boundary occurs. *symbol* must meet the requirement for symbols as described in the BNF. For a description of symbols, see subsection 4.3, Symbols.

expression
An optional absolute expression with a word-address or value attribute and with all symbols, if any, previously defined. The value of the expression must be positive. A force word boundary occurs before the expression is evaluated.

expression must meet the requirements for an expression as described in the BNF. For a description of expressions, see subsection 4.7.

The left margin of the listing shows the octal word count.

Example:

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|        |BSS       |4             |
|        |          |              |
|A       |CON       |'NAME'        |
|        |CON       |1             |
|        |CON       |2             |
|        |BSS       |16+A-W.*      |; Reserve more words so that the
|        |          |              |; total starting at A is 16
```

5.4.7 LOC - SET * COUNTER

The LOC pseudo instruction resets the location counter to the first parcel of the word address specified. The location counter is used for assigning address values to location field symbols. Changing the location counter allows code to be assembled and loaded at one location, controlled by the origin counter, then moved and executed at another address controlled by the location counter. The LOC pseudo instruction forces a word boundary within the current section before the location counter is modified.

The LOC pseudo instruction is restricted to sections that allow instructions, data, or instructions and data. If the LOC pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the LOC pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result    |Operand
|        |          |
|ignored |LOC       |[expression]
|ignored |loc       |[expression]
```

expression
Optional expression; represents the new value of the location counter. If the expression does not exist, the counter is reset to the absolute value of zero. If the expression does exist, all symbols, if any, must be previously defined. If the current base is mixed, octal is used as the base.

expression cannot have an address attribute of parcel, a relative attribute of external, or a negative value. A force word boundary occurs before the expression is evaluated.

expression must meet the requirements for an expression as described in the BNF. For a description of expressions, see subsection 4.7.

Example:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|* In this example, the code is generated and loaded at location
|* W.*+10000 and must be moved by the user to absolute location
|* 200 before execution.             |
|        |ORG       |W.*+10000        |
|        |LOC       |200              |
|LBL     |A1        |0                |
|        |.         |                 |
|        |.         |                 |
|        |.         |                 |
|        |J         |LBL              |
```

5.4.8 BITW - SET *W COUNTER

The BITW pseudo instruction resets the current bit position, relative to the bit 0 of the current word, to the value specified. A value of 64 (decimal) forces the following instruction to be assembled at the beginning of the next word (force word boundary) if the current bit position is not bit 0.

If the current bit position is bit 0, the BITW pseudo instruction does not force a word boundary, and the following instruction is assembled at bit 0 of the current word.

If the origin and location counters are set lower than their current value, any code previously generated in the overlapping portion of the word is ORed with any new code.

The BITW pseudo instruction is restricted to sections that allow data or instructions and data. If the BITW pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the BITW pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	BITW	[*expression*]
ignored	bitw	[*expression*]

expression

An optional expression. If *expression* is not specified, the default is the absolute value of 0. If *expression* is specified, it must have an address attribute of value, a relative attribute of absolute, and be a positive value in the range from 0 to 64 (decimal). All symbols within *expression*, if any, must have been previously defined. If the current base is mixed, decimal is used.

expression must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

The value generated in the code field of the listing is equal to the value of the expression.

Example:

Location	Result	Operand	Comment
1	10	20	35
	BITW	D'39	

5.4.9 BITP - SET *P COUNTER

The BITP pseudo instruction resets the bit position relative to bit 0 of the current parcel to the value specified. A value of 16 forces a parcel boundary. If the current bit position is in the middle of a parcel and a value of 16 is specified, the bit position is set to the beginning of the next parcel; otherwise, the bit position is not changed. If the origin and location counters are set lower than its current value, any code previously generated in the overlapping portion of the word is ORed with any new code.

The BITP pseudo instruction is restricted to sections that allow instructions or instructions and data. If the BITP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the BITP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	BITP	[*expression*]
ignored	bitp	[*expression*]

expression

An optional expression. If *expression* is not specified, the default is the absolute value of 0. If *expression* is specified, it must have an address attribute of value, a relative attribute of absolute, and be a positive value in the range from 0 to 16 (decimal). All symbols within *expression*, if any, must have been previously defined. If the current base is mixed, decimal is used.

expression must meet the requirements for an expression as described in the BNF. For a description of expressions, see subsection 4.7.

The value generated in the code field of the listing is equal to the value of the expression.

Examples:

Location	Result	Operand	Comment
1	10	20	35
	BITP	D'14	

In the following example, O'14 and O'12 are OR'ed and the result is 1110:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|        |vwd       |d'16/0     |; Fill first 16 bits with 0
|        |vwd       |6/o'14     |; Fill next 6 bits with 001100
|        |bitp      |0          |; Reset the pointer to bit 0
|        |          |           |; of parcel B
|        |vwd       |6/o'12     |; 001100 from the previou vwd
|        |          |           |; is OR'ed with 001010
```

Diagrams 5-1 through 5-4 illustrate what happens when CAL assembles the previous example. ↑ represents the current bit position and ^ indicates an unitialized bit in diagrams 5-1 through 5-4.

When CAL encounters the vwd d'16/0 instruction, the following is stored in parcel a:

```
|        |        |^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|
|00000000|00000000|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|
|________|________|________|________|________|________|________|________|
                  ↑
```

Diagram 5-1. BITP Example - Zeroing Parcel A

The following is stored in parcel b when vwd 6/O'14 is assembled:

```
|        |        |        |^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|
|00000000|00000000|001100^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|
|________|________|________|________|________|________|________|________|
                           ↑
```

Diagram 5-2. BITP Example - Parcel B Set by vwd Instruction

The pointer is reset to bit 0 of parcel b when the bitp 0 instruction is encountered as follows:

```
|        |        |        |^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|
|00000000|00000000|001100^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|^^^^^^^^|
|________|________|________|________|________|________|________|________|
                  ↑
```

Diagram 5-3. BITP Example - Resetting the Pointer

The next instruction, vwd 6/O'12, causes 001010 (o'12) to be OR'ed with the first six bits of parcel b (001100), producing 001110, which is stored as follows:

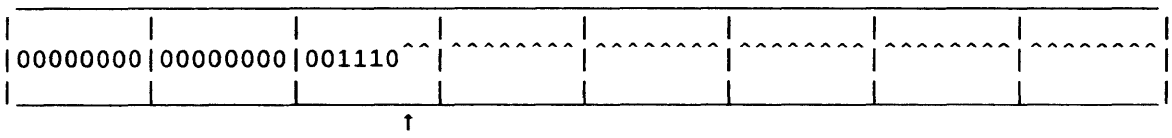

Diagram 5-4. BITP Example - Result of a Bitp Followed by a vwd

5.4.10 ALIGN - ALIGN ON AN INSTRUCTION BUFFER BOUNDARY

The ALIGN pseudo instruction ensures that the code following the instruction is aligned on an instruction buffer boundary. An offset is calculated to determine the next instruction buffer boundary from the current location counter. The size of the offset (20_8 or 40_8 words) is determined by the type of machine for which CAL is targeting code (see the cpu=*primary* option on the CAL invocation statement).

Machine Type	Octal Offset (words/parcels)
CRAY-2	20/100
CRAY X-MP	40/200
CRAY-1	20/100

The calculated offset is added to the location and origin counters within the currently enabled section. Code is not generated within this offset.

The offset is calculated relative to the beginning of a section. Each section encountering an ALIGN pseudo by means of the location counter is aligned.

If the location counter is currently positioned at an instruction buffer boundary, no alignment is performed. A warning message is issued if the section that is being aligned has a type of STACK or TASK COMMON or has a location of Local Memory.

The ALIGN pseudo instruction is restricted to sections that have a type of instruction, data, or instructions and data. If the ALIGN pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ALIGN pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*symbol*]	ALIGN	ignored
[*symbol*]	align	ignored

symbol An optional symbol; it is assigned the parcel address of the location counter after alignment.

If the optional symbol is specified in the location field, it is assigned the value of the location counter and an attribute of parcel address after alignment on the next instruction buffer boundary.

symbol must meet the requirements for symbols as described in the BNF. For a description of symbols, see subsection 4.3.

The octal value in the output listing immediately to the left of the location field indicates the number of full parcels skipped.

Example:

Location	Result	Operand	Comment
1	10	20	35
L	=	*	
	J	A	
A	ALIGN		

5.5 MESSAGE CONTROL

Two pseudo instructions, ERROR and ERRIF, allow you to generate an assembly error condition. The MLEVEL pseudo allows you to change the level of messages you receive in your source.

- ERROR Sets an assembly error flag
- ERRIF Sets an assembly error flag according to the conditions being tested
- MLEVEL Sets the level at which messages are reported in the source listing

5.5.1 ERROR - UNCONDITIONAL ERROR GENERATION

The ERROR pseudo instruction unconditionally issues a listing message. If the priority is not specified, the ERROR pseudo issues an error level message. If the condition is not satisfied (FALSE), no message is issued.

The ERROR pseudo instruction can be specified anywhere within a program segment. If the ERROR pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ERROR pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*prior*]	ERROR	ignored
[*prior*]	error	ignored

prior Optional error priority; can be one of the following classes:

The following priorities can be entered in mixed case and are directly mapped into a user-defined message of the corresponding priority:

COMMENT, NOTE, CAUTION, WARNING, or ERROR

The following priorities are mapped into a message with a priority of error:

C, D, E, F, I, L, N, O, P, R, S, T, U, V, or X

The following priorites are mapped into messages with a priority of warning:

W, W1, W2, W3, W4, W5, W6, W7, W8, W9, Y1, or Y2

Messages C through Y2 provide compatability with CAL Version 1.

CAL is capable of producing five similar messages with differing priorities (ERROR, WARNING, CAUTION, NOTE, or COMMENT). The ERROR pseudo could be used to check for valid input and to assign an appropriate message. In the following example, a user-defined message priority of ERROR is specified.

Example:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|ERROR   |ERROR     |            |; ***ERROR*** Input is invalid
```

5.5.2 ERRIF - CONDITIONAL ERROR GENERATION

The ERRIF pseudo instruction conditionally issues a listing message. If the condition is satisfied (true), the appropriate user-defined message is issued. If the priority is not specified, the ERRIF pseudo issues an error level message. If the condition is not satisfied (false), no message is issued. If any errors are encountered while evaluating the operand field, the resulting condition is handled as if true and the appropriated user-defined message is issued.

The ERRIF pseudo instruction can be specified anywhere within a program segment. If the ERRIF pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ERRIF pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result    |Operand
|        |          |
|[prior] |ERRIF     |[expression]","condition","[expression]
|[prior] |errif     |[expression]","condition","[expression]
```

prior Optional error priority; can be one of the following classes:

The following priorities can be entered in mixed case and are directly mapped into a user-defined message of the corresponding priority:

COMMENT, NOTE, CAUTION, WARNING, or ERROR

The following priorities are mapped into messages with a priority of error:

C, D, E, F, I, L, N, O, P, R, S, T, U, V, or X

The following priorites are mapped into messages with a priority of warning:

W, W1, W2, W3, W4, W5, W6, W7, W8, W9, Y1, or Y2

Messages C through Y2 provide compatability with CAL Version 1.

expression

Zero, one, or two expressions to be compared by *condition*. If one or both of the expressions are missing, a value of absolute zero is substituted for every expression that is not specified. Symbols found in either of the expressions can be defined later in a seqment.

expression must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7, Expressions.

condition

condition specifies the relationship between two expressions that causes the generation of an error. For LT, LE, GT, and GE, only the values of the expressions are examined. *condition* can be entered uppercase, lowercase, or in mixed case and can be one of the following:

condition	Significance
LT	Less than; the value of the first expression must be less than the value of the second expression.
LE	Less than or equal to; the value of the first expression must be less than or equal to the value of the second expression.
GT	Greater than; the value of the first expression must be greater than the value of the second expression.
GE	Greater than or equal to; the value of the first expression must be greater than or equal to the value of the second expression.
EQ	Equal; the value of the first expression must be equal to the value of the second expression. The expressions must both be one of the following: Absolute Immobile relative to the same section Relocatable in the program section or the same common section External relative to the same external symbol. The word-address, parcel-address or value attributes must be the same.

condition	Significance
NE	Not equal. The first expression must not equal the second expression. The expressions cannot both be absolute, or both be external relative to the same external symbol, or both be relocatable in the program section or the same common section. The word-address, parcel-address or value attributes are not the same.

The address and relative attributes are not compared by the ERRIF pseudo instruction. A CAUTION level message is issued.

Example:

Location	Result	Operand	Comment
1	10	20	35
P	ERRIF	ABC,LT,DEF	

5.5.3 MLEVEL - MESSAGE PRIORITY

The MLEVEL pseudo changes the priority of messages that you receive in your source listing. If the ML option on the CAL invocation statement differs from the option on the MLEVEL pseudo, the invocation statement overrides the pseudo.

If the option accompanying the MLEVEL pseudo is invalid, a diagnostic message is generated and MLEVEL is set to the default value.

Format:

Location	Result	Operand
ignored	MLEVEL	[*option*]
ignored	MLEVEL	*
ignored	mlevel	[*option*]
ignored	mlevel	*

The MLEVEL pseudo instruction can be specified anywhere within a program segment. If the MLEVEL pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the MLEVEL pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

option Optional message priority; can be entered in uppercase, lowercase, or mixed case and must be one of the following:

ERROR enables ERROR level messages only.

WARNING enables WARNING and ERROR level messages (default).

CAUTION enables CAUTION, WARNING, and ERROR level messages.

NOTE enables NOTE, CAUTION, WARNING, and ERROR level messages.

COMMENT enables COMMENT, NOTE, CAUTION, WARNING, and ERROR level messages.

No entry; reset to default message level.

* Reactivates the message priority in effect before the current message priority was specified within the current program segment. Each occurrence of a MLEVEL pseudo instruction other than MLEVEL * causes a new message priority to be initiated. Each MLEVEL * releases the current message priority and reactivates the message priority that preceded the current message priority. If all specified message priorities have been released when an MLEVEL * is encountered, CAL issues a message with a priority of caution and uses the default priority (WARNING).

5.5.4 DMSG - ISSUE DIAGNOSTIC MESSAGE

The DMSG pseudo issues a comment level diagnostic message containing the string found in the operand field, if a string exists. If the string contains more than 80 characters, a warning message is issued and the string is truncated.

Comment level diagnostic messages might not be issued by default on the operating system in which CAL is executing. See section 2, Binary Definition Files, for more detailed information.

The assembler recognizes up to 80 characters within the string, but the string may be truncated further when the diagnostic message is issued (depending on the operating system in which the assembler is executing).

The DMSG pseudo instruction can be specified anywhere within a program segment. If the DMSG pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the DMSG pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	DMSG	[*del-char*[*string-of-ASCII*]*del-char*]
ignored	dmsg	[*del-char*[*string-of-ASCII*]*del-char*]

del-char Delimiter character; must be a single matching character on both ends of the ASCII character string. A character string can be delimited by a character other than an apostrophe. Any ASCII character other than a space can be used. Two consecutive occurrences of the delimiting character indicate a single such character is to be included in the character string.

string-of-ASCII
An ASCII character string to be printed as the main title on subsequent pages of the listing. A maximum of 80 characters is allowed.

NOTE

Using the DMSG pseudo for assembly timings may be deceiving. For example, if the DMSG pseudo is inserted near the beginning of an assembler segment, more time may elapse (from the time that CAL begins assembling the segment to the time the message is issued) than you may have expected.

5.6 LISTING CONTROL

Listing control pseudo instructions allow you to control the content and format of the listing produced by the assembler. The listing control pseudo instructions are as follows: LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, and ENDTEXT. These pseudo instructions are not listed unless the LIST pseudo instruction is specified with the LIS option.

- LIST Controls listing by specifying particular listing features to be enabled or disabled
- SPACE Blank lines may be inserted in listing
- EJECT Begin new page
- TITLE Main title printed on each of listing
- SUBTITLE Subtitle printed on each page of listing
- TEXT Declare beginning of global text source
- ENDTEXT Terminate global text source

5.6.1 LIST - LIST CONTROL

The LIST pseudo instruction controls the listing. LIST is a list control pseudo and is, by default, not listed. To include the LIST pseudo on the listing, specify the LIS option on this instruction. An END pseudo instruction causes options to be reset to the default values.

The LIST pseudo instruction can be specified anywhere within a program segment. If the LIST pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the LIST pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*name*]	LIST	[*option*]{","[*option*]}
[*name*]	LIST	*
[*name*]	list	[*option*]{","[*option*]}
[*name*]	list	*

name Optional list name. *name* must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2, Names.

If *name* is present, the instruction is ignored unless a matching name is specified on the list parameter on the CAL invocation statement. LIST pseudos with a matching name are not ignored. LIST pseudos with a blank location field are always processed.

[*option*]{","[*option*]}

All of the option names given below can be specified in some form as CAL invocation statement parameters. The selection of an option on the CAL invocation statement overrides the enabling or disabling of the corresponding feature by a LIST pseudo.

If the no list option is used on the CAL invocation statement, all LIST pseudos in the program are processed.

Listing option. Specifies that a particular listing feature be enabled or disabled. There can be zero, one, or more options specified or an *. The options allowed are listed below. Defaults are underlined. If no options are specified, OFF is assumed.

ON OFF	ON	Enable source statement listing. Source statements and code generated are listed.
	OFF or blank operand field	Disable source statement listing. Only statements with errors are listed while this option is selected. Listing control pseudo instructions are also listed if LIS option is enabled.
ED NED	ED	Enable listing of edited statements
	NED	Disable listing of edited statements
XRF NXRF	XRF	Enable cross-reference. Symbol references are accumulated and a cross-reference listing is produced.
	NXRF	Disable cross-reference. Symbol references are not accumulated. If this option is selected when the END pseudo is encountered, no cross-reference is produced.
XNS NXNS	XNS	Include nonreferenced local symbols in the reference. Local symbols that were not referenced in the listing output are included in the cross-reference listing.
	NXNS	Exclude nonreferenced local symbols in the cross-reference. If this option is selected when the END pseudo is encountered, local symbols that were not referenced in the listing output are not included in the cross-reference.

LIS
NLIS

LIS Enable listing of the pseudo instructions LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, and ENDTEXT.

NLIS Disable listing of these pseudo instructions.

TXT
NTXT

TXT Enable global text source listing. Each statement following a TEXT pseudo instruction is listed through the ENDTEXT instruction, if the listing is otherwise enabled.

NTXT Disable global text source listing. Statements following a TEXT pseudo instruction through the following ENDTEXT instruction are not listed.

MAC
NMAC

MAC Enable listing of macro and opdef expansions. Statements generated by macro and opdef calls are listed. Conditional statements and skipped statements generated by macro and opdef calls are not listed unless the macro conditional list feature is enabled (MIF).

NMAC Disable listing of macro and opdef expansions. Statements generated by macro and opdef calls are not listed.

MBO
NMBO

MBO Enable listing of generated statements before editing. Only statements that produce generated code are listed. The listing of macro expansions (MAC) or the listing of duplicated statements (DUP) must also be enabled.

NMBO Disable listing of statements that produce generated code. Statements generated by a macro or opdef call (MAC), or by a DUP or ECHO (DUP) pseudo instruction, are not listed before editing.

MIC
NMIC

Source statements containing a micro reference or a concatenation character are listed before editing regardless of whether this option is enabled or disabled.

MIC Enable listing of generated statements before editing. Statements which are generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, and which contain a micro reference or concatenation character are listed before and after editing. The listing of macro expansions or the listing of duplicated statements must also be enabled.

NMIC Disable listing of generated statements before editing. Statements generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, are not listed before editing.

MIF
NMIF — Conditional statements and skipped statements in source code are listed regardless of whether this option is enabled or disabled.

MIF Enable macro conditional listing. Conditional statements and skipped statements generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, are listed. The listing of macro expansions or the listing of duplicated statements must also be enabled.

NMIF Disable macro conditional listing. Conditional statements and skipped statements generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction are not listed.

DUP
NDUP

DUP Enable listing of duplicated statements. Statements generated by DUP and ECHO expansions are listed. Conditional statements and skipped statements generated by DUP and ECHO are not listed unless the macro conditional list feature is enabled (MIF).

NDUP Disable listing of duplicated statements. Statements generated by DUP and ECHO are not listed.

* Reactivates the LIST pseudo in effect before the current LIST pseudo was specified within the current program segment. Each occurrence of a LIST pseudo instruction other than LIST * causes a new listing control to be initiated. Each LIST * releases the current listing control and reactivates the listing control that preceded the current list control. If all specified listing controls have been released when a LIST * is encountered, CAL issues a message with a priority of CAUTION and uses the defaults for listing control.

5.6.2 SPACE - LIST BLANK LINES

The SPACE pseudo instruction inserts the number of blank lines specified into the output listing. SPACE is a list control pseudo instruction and is, by default, not listed. To include the SPACE pseudo on the listing, specify the LIS option on the LIST pseudo instruction.

The SPACE pseudo instruction can be specified anywhere within a program segment. If the SPACE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SPACE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	SPACE	[*expression*]
ignored	space	[*expression*]

expression

An optional absolute expression specifying the number of blank lines to insert in the listing. *expression* must have an address attribute of value, a relative attribute of absolute, and a value of 0 or greater.

If *expression* is not specified, the absolute value of one is used and one blank line is inserted into the output listing. If the current base is mixed, a default of decimal is used for the expression.

expression must meet the requirement for an expression as described in the BNF. For a description of expressions, see subsection 4.7.

5.6.3 EJECT - BEGIN NEW PAGE

The EJECT pseudo instruction causes a page eject on the output listing. EJECT is a list control pseudo and is, by default, not listed. To include the EJECT pseudo on the listing, specify the LIS option on the LIST pseudo instruction.

The EJECT pseudo instruction can be specified anywhere within a program segment. If the EJECT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the EJECT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	EJECT	ignored
ignored	eject	ignored

5.6.4 TITLE - SPECIFY LISTING TITLE

The TITLE pseudo instruction specifies the main title to be printed on the listing. TITLE is a list control pseudo and is, by default, not listed. To include the TITLE pseudo on the listing, specify the LIS option on the LIST pseudo instruction.

The TITLE pseudo instruction can be specified anywhere within a program segment. If the TITLE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the TITLE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	TITLE	*del-char*[*string-of-ASCII*]*del-char*
ignored	title	*del-char*[*string-of-ASCII*]*del-char*

del-char Delimiter character; must be a single matching character on both ends of the ASCII character string. A character string can be delimited by a character other than an apostrophe. Any ASCII character other than a space can be used. Two consecutive occurrences of the delimiting character indicate a single such character is to be included in the character string.

string-of-ASCII
An ASCII character string to be printed as the main title on subsequent pages of the listing. A maximum of 72 characters is allowed.

5.6.5 SUBTITLE - SPECIFY LISTING SUBTITLE

The SUBTITLE pseudo instruction specifies the subtitle to be printed on the listing. The instruction also causes a page eject. SUBTITLE is a list control pseudo and is, by default, not listed. To include the SUBTITLE pseudo on the listing, specify the LIS option on the LIST pseudo instruction.

The SUBTITLE pseudo instruction can be specified anywhere within a program segment. If the SUBTITLE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SUBTITLE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	SUBTITLE	*del-char*[*string-of-ASCII*]*del-char*
ignored	subtitle	*del-char*[*string-of-ASCII*]*del-char*

del-char Delimiter character; must be a single matching character on both ends of the ASCII character string. A character string can be delimited by a character other than an apostrophe. Any ASCII character other than a space can be used. Two consecutive occurrences of the delimiting character indicate a single such character is to be included in the character string.

string-of-ASCII
An ASCII character string to be printed as the main title on subsequent pages of the listing.

5.6.6 TEXT - DECLARE BEGINNING OF GLOBAL TEXT SOURCE

Source lines following the TEXT pseudo instruction through the next ENDTEXT pseudo instruction are treated as *text* source statements. These statements are listed only when the TXT listing option is enabled. A symbol defined in *text* source is treated as a text symbol for cross-reference purposes. That is, such a symbol is not listed in the cross-reference unless there is a reference to the symbol from a listed statement. The text *name* part of the cross-reference listing contains the text name.

Symbols defined in *text* source are global if the text appears in the global part of a program segment. Symbols in *text* source are local if the text appears within a program module.

TEXT is a list control pseudo instruction and is, by default, not listed. The TEXT pseudo is listed if the listing is on or if the LIS listing option is enabled regardless of other listing options.

The TEXT and ENDTEXT pseudo instructions have no effect on a binary definition file.

The TEXT pseudo instruction can be specified anywhere within a program segment. If the TEXT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the TEXT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*name*]	TEXT	[*del-char*[*string-of-ASCII*]*del-char*]
[*name*]	text	[*del-char*[*string-of-ASCII*]*del-char*]

name Optional name of text. *name* is used as the name of the source following until the next ENDTEXT pseudo instruction. The name found in the location field is the text name for all defined symbols in the section, and is listed in the text name part of the cross reference listing.

name must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

del-char Delimiter character; must be a single matching character on both ends of the ASCII character string. A character string can be delimited by a character other than an apostrophe. Any ASCII character other than a space can be used. Two consecutive occurrences of the delimiting character indicate a single such character is to be included in the character string.

string-of-ASCII
An ASCII character string to be printed as the main title on subsequent pages of the listing. A maximum of 72 characters is allowed.

5.6.7 ENDTEXT - TERMINATE GLOBAL TEXT SOURCE

The ENDTEXT pseudo instruction terminates *text* source initiated by a TEXT instruction. An IDENT or END pseudo instruction also terminates *text* source.

The ENDTEXT is a list control pseudo and by default is not listed unless the TXT option is enabled. If the LIS option is enabled, the ENDTEXT instruction is listed regardless of other listing options.

The ENDTEXT pseudo instruction can be specified anywhere within a program segment. If the ENDTEXT pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ENDTEXT pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	ENDTEXT	ignored
ignored	endtext	ignored

Example (with TXT option off):

Source listing:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	TEXT	
A	=	2	
TXTNAME	TEXT	'An example.'	
B	=	3	
C	=	4	
	ENDTEXT		
	A1	A	
	A2	B	
	END		

Output listing:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	TEXT	
A	=	2	
TXTNAME	TEXT	'An example.'	
	A1	A	
	A2	B	
	END		

5.7 SYMBOL DEFINITION

The pseudo instructions =, SET, and MICSIZE define symbols used in the program.

Requirements for symbols are given in subsection 4.3.

- = Equates a symbol to a value; not redefinable.
- SET Sets a symbol to a value; redefinable.
- MICSIZE Equates a symbol to a value equal to the number of characters in micro string; redefinable.

5.7.1 = - EQUATE SYMBOL

The = pseudo instruction defines a symbol with the value and attributes determined by the expression. The symbol is not redefinable.

The = pseudo instruction can be specified anywhere within a program segment. If the = pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the = pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*symbol*]	=	*expression*["," [*attribute*]]

symbol An optional unqualified symbol. The symbol is implicitly qualified by the current qualifier. The symbol must not be defined already. The location field can be blank.

symbol must satisfy the requirements for symbols as described in the BNF. For a description of symbols, see subsection 4.3.

expression
All symbols found within *expression* must have been previously defined.

expression must meet the requirements for an expression as described in the BNF. For a desciption of expressions, see subsection 4.7.

attribute *An optional* P, W, or V indicating parcel, word, or value attribute. Attribute, if present, is used instead of the expression's attribute. An expression with word-address attribute is multiplied by four if a parcel-address attribute is specified; an expression with parcel-address attribute is divided by four if word-address attribute is specified. A relocatable expression cannot be specified as having value attribute.

Example:

Location	Result	Operand	Comment
1	10	20	35
SYMB	=	A*B+100/4	

5.7.2 SET - SET SYMBOL

The SET pseudo instruction resembles the = pseudo instruction. However, a symbol defined by SET is redefinable.

The SET pseudo instruction can be specified anywhere within a program segment. If the SET pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SET pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*symbol*]	SET	*expression*["," [*attribute*]]
[*symbol*]	set	*expression*["," [*attribute*]]

symbol Optional unqualified symbol. The symbol is implicitly qualified by the current qualifier. The symbol must not be defined already. The location field can be blank.

symbol must meet the requirements for symbols as described in the BNF. For a description of symbols, see subsection 4.3.

expression

All symbols found within *expression* must have been previously defined.

expression must meet the requirements for an expression as described in the BNF. For a description of expressions, see subsection 4.7.

attribute

An optional P, W, or V indicating parcel, word, or value attribute. Attribute, if present, is used instead of the expression's attribute. An expression with word-address attribute is multiplied by four if a parcel-address attribute is specified; an expression with parcel-address attribute is divided by four if word-address attribute is specified. An immobile or relocatable expression cannot be specified as having a value attribute.

Example:

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|SIZE    |=         |O'100         |
|PARAM   |SET       |D'18          |
|WORD    |SET       |*W            |
|PARCEL  |SET       |*P            |
|SIZE    |=         |SIZE+1        |(Illegal)
|PARAM   |SET       |PARAM+2       |(Legal)
```

5.7.3 MICSIZE - SET REDEFINABLE SYMBOL TO MICRO SIZE

The MICSIZE pseudo instruction defines the symbol in the location field as a symbol with an address attribute of value, a relative attribute of absolute, and a value equal to the number of characters in the micro string whose name is in the operand field. Another SET or MICSIZE instruction with the same symbol redefines the symbol to a new value.

The MICSIZE pseudo instruction can be specified anywhere within a program segment. If the MICSIZE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the MICSIZE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*symbol*]	MICSIZE	*name*
[*symbol*]	micsize	*name*

symbol Optional unqualified symbol. *symbol* is implicitly qualified by the current qualifier. The location field can be blank.

symbol must meet the requirement for a symbol as described in the BNF. For a description of symbols, see subsection 4.3.

name The name of a micro string previously defined. *name* must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

5.8 DATA DEFINITION

Data definition instructions are the only pseudo instructions that generate object binary. The only other instructions that are translated into object binary are the symbolic machine instructions. An instruction that generates binary cannot be used with a section that does not allow instructions, data, or instructions and data.

- CON Places an expression value into one or more words
- BSSZ Generates words that have been initialized to zero
- DATA Generates one or more words of numeric or character data
- VWD Generates a variable-width field of word-oriented data

5.8.1 CON - GENERATE CONSTANT

The CON pseudo instruction generates one or more full words of binary data. This pseudo always causes a force word boundary.

The CON pseudo instruction is restricted to sections that have a type of data or instructions and data. If the CON pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the CON pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*symbol*]	CON	[*expression*]{","[*expression*]}
[*symbol*]	con	[*expression*]{","[*expression*]}

symbol An optional symbol; assigned the word address value of the location counter after the force word boundary occurs. *symbol* must meet the requirements for a symbol as described in the BNF. For a description of symbols, see subsection 4.3.

expression
An expression whose value is to be inserted into a single 64-bit word. If an expression is null, a single zero word is generated. A force word boundary occurs before any operand field expressions are evaluated. A double-precision, floating-point constant is not allowed.

expression must meet the requirements for an expression as described in the BNF. For a description of expressions, see subsection 4.7.

Example:

Location	Result	Operand	Comment
1	10	20	35
A	CON	O'7777017	
	CON	A	; Generates the address of A

5.8.2 BSSZ - GENERATE ZEROED BLOCK

The BSSZ pseudo instruction causes a block of words containing zeros to be generated. When BSSZ is specified, a force word boundary occurs, and the number of zero words specified by the operand field expression is generated.

The BSSZ pseudo instruction is restricted to sections that have a type of data or instructions and data. If the BSSZ pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the BSSZ pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*symbol*]	BSSZ	[*expression*]
[*symbol*]	bssz	[*expression*]

symbol Optional symbol; assigned the word-address value of the location counter after the force word boundary occurs. *symbol* must meet the requirements for a symbol as described in the BNF. For a description of symbols, see subsection 4.3.

expression
An optional absolute expression with an attribute of word address or value and with all symbols previously defined. The expression value must be positive and specifies the number of 64-bit words containing zeros to be generated. A blank operand field results in no data generation.

expression must meet the requirement for an expression as described in the BNF. For a description of expressions, see subsection 4.3.

The left margin of the listing shows the octal word count of a BSSZ.

5.8.3 DATA - GENERATE DATA WORDS

The DATA pseudo instruction generates zero or more bits of code for each data item parameter found in the operand field. If a label exists in the location field, a force word boundary occurs and the symbol is assigned an address attribute and the value of the current location counter.

If a label is not included in the location field, a force word boundary does not occur.

The DATA pseudo instruction is restricted to sections that have a type of data or instructions and data. If the DATA pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the DATA pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The length of the field generated for each data item depends on the type of constant involved. Data-items produce zero or more bits of absolute value binary code as follows:

Data-item	Description
Floating	One or two binary words, depending on whether the data item is a single- or double-precision data item
Integer	One binary word
Character	Zero or more bits of binary code depending on the following: Character set specified Number of characters in the string Character count (optional) Character suffix (optional)

A word boundary is not forced between data items.

Format:

Location	Result	Operand
[*symbol*]	DATA	[*data-item*]{","[*data-item*]}
[*symbol*]	data	[*data-item*]{","[*data-item*]}

symbol Optional symbol assigned the word address of the location counter after a force word boundary. If no symbol is present, a force word boundary does not occur.

symbol must meet the requirements for a symbol as described in the BNF. For a description of symbols, see subsection 4.3.

data-item A numeric or character data item. *data-item* must meet the requirements for a data item as described in the BNF. For a description of data items, see subsection 4.4, Data.

The DATA pseudo works with the actual number of bits given in the data item.

Examples:

1. Unlabeled data items are stored in the next available bit position.

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|        |IDENT     |EXDAT         |
|        |DATA      |'abcd'*       |; Unlabeled data item 1
|        |DATA      |'efgh'*       |; Unlabeled data item 2
|        |END       |              |
```

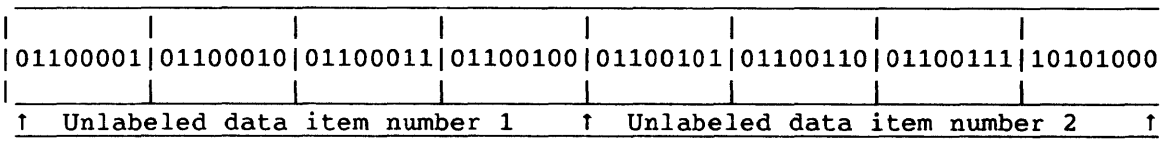

Diagram 5-5. Storage of Unlabeled Data Items

2. Labeled data items cause a force word boundary.

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|        |IDENT     |EXDAT         |
|        |DATA      |'abcd'*       |; Unlabeled data item 1
|ALPHA   |DATA      |'efgh'*       |; Labeled data item 1
|BETA    |DATA      |'ijkl'*       |; Labeled data item 2
|        |DATA      |'mnop'*       |; Unlabeled data item 2
```

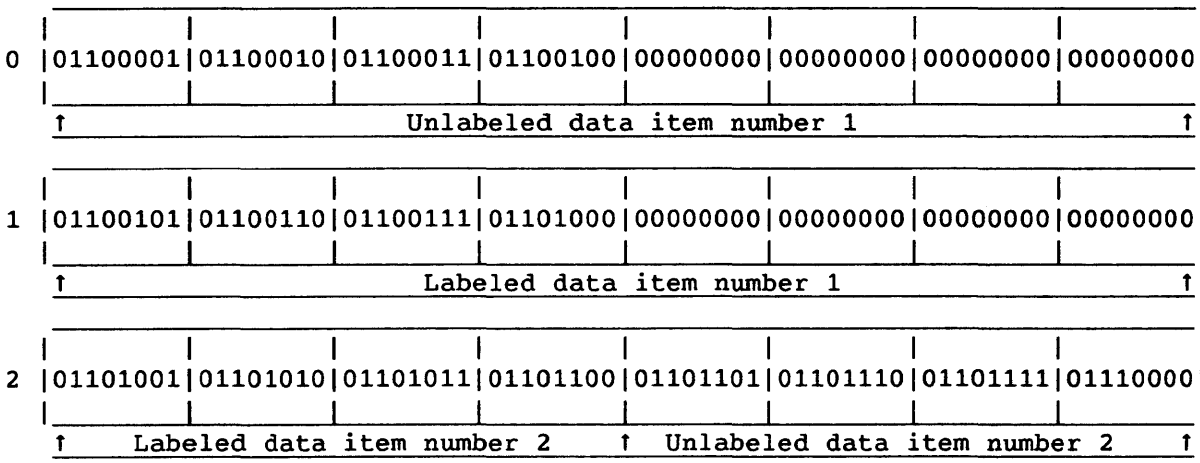

Diagram 5-6. Storage of Labeled and Unlabeled Data Items

3. Data is stored bit by bit in consecutive words, if no force word boundary occurs. Note: The following data-item is defined with the CDC character set (6 bits per character).

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |IDENT     |EXDAT            |
|        |DATA      |C'ABCDEFGHIJK'*  |; Unlabeled data item 1
|        |END       |                 |
```

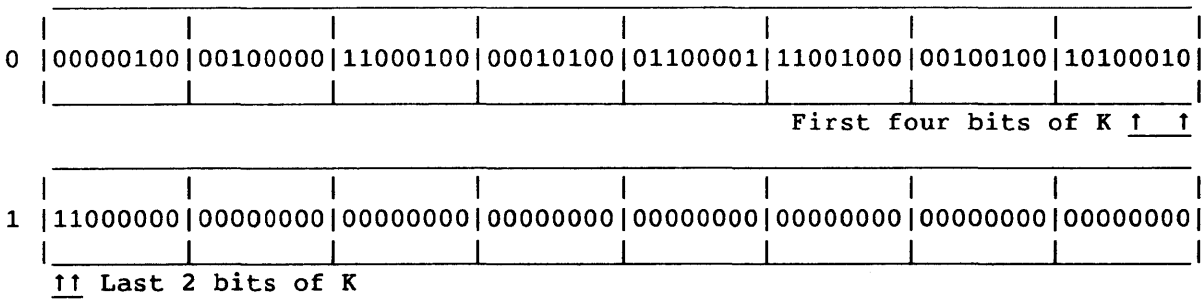

Diagram 5-7. Storage of CDC Character Data Item

Example:

```
|Code generated            |Location|Result    |Operand          |Comment
|                          |1       |10        |20               |35
|                          |        |          |                 |
|                          |        |IDENT     |EXAMPLE          |
|00000000000000005252      |        |DATA      |O'5252,A'ABC'R   |
|00000000000020241103      |        |          |                 |
|0405022064204010020040    |        |DATA      |'ABCD'           |
|0425062164404010020040    |        |DATA      |'EFGH'           |
|040502206420              |        |DATA      |'ABCD'*          |
|          10521443510     |        |DATA      |'EFGH'*          |
|00000000000000000000      |        |DATA      |'ABCD'12R        |
|040502206420              |        |          |                 |
|          10521443510     |        |DATA      |'EFGHIJ'*        |
|044512                    |        |          |                 |
|0405022064204010020040    |LL2     |DATA      |'ABCD'           |
|                          |        |          |                 |
|                          |        |          |                 |
|00000000000000000144      |        |DATA      |100              |
|037743527461670430 2142   |        |DATA      |1.25E-9          |
|                          |        |          |                 |
|                          |        |          |                 |
|0521102225144022251440    |        |DATA      |'THIS IS A MESSAGE'*L
|0404402324252324640507    |        |          |                 |
|0424                      |        |          |                 |
|  000                     |        |VWD       |8/0              |
|                          |        |END       |                 |
```

5.8.4 VWD - VARIABLE WORD DEFINITION

The VWD pseudo instruction allows data to be generated in fields from 0 to 64 bits wide. Fields can cross word boundaries. Data begins at the current bit position unless a symbol is used. If a symbol is used, a force word boundary occurs and the data begins at the new current bit position.

Code for each subfield is packed tightly with no unused bits inserted.

The VWD pseudo instruction is restricted to sections that have a type of instructions, data, or instructions and data. If the VWD pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the VWD pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*symbol*]	VWD	[*count*"/"[*expression*]]{","[*count*"/"[*expression*]]}
[*symbol*]	vwd	[*count*"/"[*expression*]]{","[*count*"/"[*expression*]]}

symbol Optional symbol. If present, a force word boundary occurs. The symbol is defined with the value of the location counter after the force word boundary and has an address attribute of word.

symbol must meet the requirements for symbols as described in the BNF. For a description of symbols, see subsection 4.3.

count Field width, specifying the number of bits in the field. A numeric constant or symbol, with absolute and value attributes. *count* must be positive and less than or equal to 64.

If *symbol* is specified for count, it must have been previously defined. If one or more *count/* entries are invalid, no code is generated for the entire set of subfields in the operand field. Each subfield is still evaluated, however.

expression

An expression whose value is to be inserted in the field. If *expression* is missing, the absolute value of zero is used. If *count* is not equal to zero, the count is the number of bits reserved to store the following expression, if any.

expression must meet the requirement for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

Example:

In the following example, the value of SIGN is 1, the value of FC is 0, the value of ADD is 653 (octal), and the value of DSN is $IN in ASCII code.

```
|Code generated          |Location|Result  |Operand          |Comment
|                        |1       |10      |20               |35
|                        |        |        |                 |
|                        |        |BASE    |M                |
|                        |PDT     |BSS     |0                |
|10000000000000023440515 |        |VWD     |1/SIGN,3/0,60/A'"NAM"'R
|10000000653             |        |VWD     |1/1,6/FC,24/ADD  |
|                    41  |REMDR   |=       |64-*W            |
|          00011044516   |        |VWD     |REMDR/DSN        |
```

5.9 CONDITIONAL ASSEMBLY

The instructions described in this section permit optional assembly or skipping of source code. The conditional pseudo instructions IFA, IFC, or IFE determine whether the sequence of instructions following the test is to be skipped or assembled. The end of the conditional sequence is determined by a count of instructions provided on the test instruction or by an ENDIF pseudo instruction with a matching location field name.

The ELSE pseudo instruction provides a means of reversing the effect of a previous IFA, IFE, IFC, SKIP, or ELSE instruction. The SKIP pseudo instruction unconditionally skips following statements.

When skipping under the control of a statement count, comment statements (asterisk in column 1) and continued lines are not included in the statement count.

When skipping is initiated by an IFA, IFE, IFC, SKIP, or ELSE pseudo instruction, editing is disabled. When the skip sequence has been completed, the assembler returns to the editing mode in effect before skipping was initiated.

To specify a conditional assembly, use the following pseudo instructions:

- IFA Tests expression attributes; address and relative attributes.
- IFE Tests two expressions for some assembly condition; less than, greater than, equal to.
- IFC Tests two character strings for assembly condition; less than, greater than, equal to.
- SKIP Unconditionally skip subsequent statements.
- ENDIF Terminates conditional code sequence.
- ELSE Reverses assembly condition.

5.9.1 IFA - TEST EXPRESSION ATTRIBUTE FOR ASSEMBLY CONDITION

The IFA pseudo instruction tests an attribute of an expression. If the expression has the specified attribute, assembly continues with the next statement. If the result of the attribute test is false, subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

If any errors are encountered while evaluating the attribute-condition, the resulting condition is handled as if true and the appropriate listing message is issued.

The IFA pseudo instruction can be specified anywhere within a program segment. If the IFA pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the IFA pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*name*]	IFA	["#"]*exp-attribute*,"*expression*[,"[*count*]]
[*name*]	IFA	["#"]*redef-attribute*,"*symbol*[,"[*count*]]
[*name*]	IFA	["#"]*reg-attribute*,"*reg-arg-value*[,"[*count*]]
[*name*]	IFA	["#"]*micro-attribute*,"*mname*[,"[*count*]]
[*name*]	ifa	["#"]*exp-attribute*,"*expression*[,"[*count*]]
[*name*]	ifa	["#"]*redef-attribute*,"*symbol*[,"[*count*]]
[*name*]	ifa	["#"]*reg-attribute*,"*reg-arg-value*[,"[*count*]]
[*name*]	ifa	["#"]*micro-attribute*,"*mname*[,"[*count*]]

name Optional name of conditional sequence of code. A conditional sequence of code controlled by a *name* is ended by an ENDIF pseudo with a matching name. The condition of a conditional sequence of code controlled by a *name* can be reversed by an ELSE pseudo with a matching name. If both *name* and *count* are present, *name* takes precedence.

name must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

"#" The optional # negates the condition. If errors occur in the attribute condition, the condition is evaluated as if it were true. While the # does not change the condition, it does specify the "if not" condition.

exp-attribute","*expression*

Expression attribute; *exp-attribute* is a mnemonic signifying an attribute of an expression. An expression has only one address attribute (VAL, PA, or WA) and relative attribute (ABS, IMM, REL, or EXT).

An attribute can also be any of the following mnemonics preceded by a complement sign (#) indicating that the second subfield does not satisfy the corresponding condition. All of the following mnemonics can be specified in mixed case.

Mnemonic	Attribute
VAL	Value; requires all symbols, if any, within the expression to be previously defined.
PA	Parcel address; requires all symbols, if any, within the expression to be previously defined.
WA	Word address; requires all symbols, if any, within the expression to be previously defined.
ABS	Absolute; requires all symbols, if any, within the expression to be previously defined.
IMM	Immobile; requires all symbols, if any, within the expression to be previously defined.
REL	Relocatable; requires all symbols, if any, within the expression to be previously defined.
EXT	External; requires all symbols, if any, within the expression to be previously defined.
CODE	Immobile or relocatable; relative to a code section. CODE requires all symbols, if any, within the expression to be previously defined.
DATA	Immobile or relocatable; relative to a data section. DATA requires all symbols, if any, within the expression to be previously defined.
MIXED	Immobile or relocatable; relative to a common section. MIXED requires all symbols, if any, within the expression to be previously defined.
COM	Immobile or relocatable; relative to a common section. COM requires all symbols, if any, within the expression to be previously defined.

Mnemonic	Attribute
COMMON	Immobile or relocatable; relative to a common section. COMMON requires all symbols, if any, within the expression to be previously defined.
TASKCOM	Immobile or relocatable; relative to a task common section. TASKCOM requires all symbols, if any, within the expression to be previously defined.
DYNAMIC	Immobile or relocatable; relative to a dynamic section. DYNAMIC requires all symbols, if any, within the expression to be previously defined.
STACK	Immobile or relocatable; relative to a stack section. STACK requires all symbols, if any, within the expression to be previously defined.
CM	Immobile or relocatable; relative to a section that is placed into common memory. CM requires all symbols, if any, within the expression to be previously defined.
EM	Immobile or relocatable; relative to a section that is placed into extended memory. EM requires all symbols, if any, within the expression to be previously defined. If EM is specified for a Cray Computer System other than a CRAY X-MP Computer System, the condition always fails.
LM	Immobile or relocatable; relative to a section that is placed into local memory. LM requires all symbols, if any, within the expression to be previously defined. If LM is specified for Cray Computer System other than a CRAY-2 Computer System, the condition always fails.
DEF	True if all symbols in the expression have been previously defined, otherwise the condition is false.
	expression must meet the requirement for an expression as described in the BNF. For a description of expressions, see subsection 4.7.

redef-attribute","*symbol*

Redefinable attribute; the condition is true if the symbol following *redef-attribute* is redefinable; otherwise, the condition is false.

Mnemonic	Attribute
SET	The symbol in the second subfield is a redefinable symbol.

symbol must meet the requirements for a symbol as described in the BNF. For a description of symbols, see subsection 4.3.

reg-attribute","*reg-arg-value*

Register-attribute. If REG is specified, the condition is true if the following string is a valid complex-register; otherwise, the condition is false. Register-attribute is defined as follows:

Mnemonic	Attribute
REG	The second subfield contains a valid A, B, S, T, or V register designator.

reg-arg-value is any ASCII character up to but not including a legal terminator (blank character or semicolon; new format) and element separator character (,).

micro-attribute","*mname*

If MIC is specified, the condition is true if the following identifier is an existing micro name; otherwise, the condition is false. *micro-attribute* is defined as follows:

Mnemonic	Attribute
MIC	The name in the second subfield is a micro name.

mname must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

count Statement count; must be an absolute expression with positive value. All symbols in the expression, if any, must be previously defined. A missing or null count subfield gives a zero count.

count is only used when the location field is not specified. If *name* is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

Example:

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|SYM1    |SET       |1             |
|SYM2    |=         |2             |
|        |IFA       |SET,SYM1,2    |; If the condition is true,
|        |S1        |SYM1          |; include this statement
|        |S2        |SYM2          |; include this statement
|SYM2    |=         |1             |
|        |IFA       |SET,SYM2,1    |; If the condition is false,
|        |S3        |SYM2          |; skip this statement
```

5.9.2 IFC - TEST CHARACTER STRINGS FOR ASSEMBLY CONDITION

The IFC pseudo instruction tests a pair of character strings for a condition under which code is to be assembled if the relation specified by *condition* is satisfied (true). If the relationship is not satisfied (false), subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

If any errors are encountered while evaluating the string condition, the resulting condition is handled as if true and an appropriate listing message is issued.

The IFC pseudo instruction can be specified anywhere within a program segment. If the IFC pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the IFC pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result    |Operand
|        |          |
|[name]  |IFC       |[string]","condition","[string][","[count]]
|[name]  |ifc       |[string]","condition","[string][","[count]]
```

name Optional name of a conditional sequence of code. A conditional sequence of code controlled by a *name* is ended by an ENDIF pseudo with a matching name. The condition of a conditional sequence of code controlled by a *name* can be reversed by an ELSE pseudo with a matching name. If both *name* and *count* are present, *name* takes precedence.

name must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

string Character strings to be compared. The first and third subfields can be null (empty) indicating a null character string.

The ASCII character code value of each character in the first string is compared with the value of each character in the second string. The comparison is from left to right and continues until an inequality is found or until the longer string is exhausted. A zero value is substituted for missing characters in the shorter string. Micros and formal parameters can be contained in the character strings.

string is an optional ASCII character string that must be specified with a single matching character on both ends. A character string can be delimited by any ASCII character other than a comma or space. Two consecutive occurrences of the delimiting character indicate a single such character is to be included in the character string. For example,

```
AIF IFC =O'100=,EQ,*ABCD***
```

compares the character strings O'100 and ABCD*.

condition Specifies the relation to be satisfied by the two strings. *condition* can be entered in mixed case and must be one of the following:

condition	Description
LT	Less than; the value of the first string must be less than the value of the second string.
LE	Less than or equal to; the value of first string must be less than or equal to the second string.

condition	Description
GT	Greater than; the value of first string must be greater than the value of the second string.
GE	Greater than or equal to; the value of first string must be greater than or equal to the second string.
EQ	Equal; the value of first string must be equal to the value of the second string.
NE	Not equal; the value of the first string must not equal the value of the second string.

count Statement count; must be an absolute expression with positive value. All symbols in the expression, if any, must be previously defined. A missing or null count subfield gives a zero count.

count is only used when the location field is not specified. If *name* is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

Examples:

1. In the following example, the first string is delimited by the @, and the second string is delimited by %. The first string is equal to the second string.

```
|Location|Result    |Operand              |Comment
|1       |10        |20                   |35
|        |          |                     |
|        |IDENT     |TEST                 |
|EX1     |IFC       |@ABC@@D@,EQ,%ABC@D% ; The condition is true;
|        |          |                     |; skipping does not occur.
|        |S1        |1                    |; Statement is included.
|        |S2        |2                    |; Statement is included.
|EX1     |ELSE      |                     |; Statements within the ELSE
|        |          |                     |; sequence are included only if
|        |          |                     |; the condition fails
|        |S3        |3                    |; Statement is skipped.
|EX1     |ENDIF     |                     |; End of skip sequence EX1
|        |          |                     |
|        |END       |                     |
```

2. In the following example, the first string is not equal to the second string, the two statements following the IFC are skipped.

```
|Location|Result    |Operand              |Comment
|1       |10        |20                   |35
|        |          |                     |
|        |IDENT     |TEST                 |
|EX1     |IFC       |@ABBCD@,EQ,@ABCD@ ; The condition is false;
|        |          |                     |; skipping occurs
|        |S1        |1                    |; This statement is skipped
|        |S2        |2                    |; This statement is skipped
|EX1     |ENDIF     |                     |; End of skip sequence
|        |S3        |3                    |; This statement is included
|        |          |                     |; irregardless of whether the
|        |          |                     |; condition is true or false
|        |END       |                     |
```

5.9.3 IFE - TEST EXPRESSIONS FOR ASSEMBLY CONDITION

The IFE pseudo instruction tests a pair of expressions for a condition. Code is assembled if the relation (*condition*) specified by the operation is satisfied. If the relationship between the expressions is true, assembly resumes with the next statement. If the relationship between the expressions is false, subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

If any errors are encountered while evaluating the expression-condition, the resulting condition is handled as if true and an appropriate listing message is issued.

If an assembly error is detected, assembly continues with the next statement.

The IFE pseudo instruction can be specified anywhere within a program segment. If the IFE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the IFE pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result    |Operand
|        |          |
|[name]  |IFE       |[expression]","condition","[expression][","[count]]
|[name]  |ife       |[expression]","condition","[expression][","[count]]
```

name Optional name of a conditional sequence of code. A conditional sequence of code controlled by a *name* is ended by an ENDIF pseudo with a matching name. The condition of a conditional sequence of code controlled by a *name* can be reversed by an ELSE pseudo with a matching name. If both *name* and *count* are present, *name* takes precedence.

name must meet the requirements for identifiers as described in the BNF. For a description of names, subsection 4.2.

expression

Expressions to be compared. All symbols in the expression must be previously defined. If an expression is not specified, the absolute value of zero is used.

Expressions must meet the requirements for expressions as described the BNF. For a description of expressions, see subsection 4.7.

condition Specifies the relation to be satisfied by the two strings. *condition* can be entered in mixed case and must be one of the following:

condition	Description
LT	Less than; the value of the first expression must be less than the value of the second expression; the attributes are not checked.
LE	Less than or equal to; the value of first expression must be less than or equal to the second expression; the attributes are not checked.
GT	Greater than; the value of first expression must be greater than the value of the second expression; the attributes are not checked.
GE	Greater than or equal to; the value of first expression must be greater than or equal to the second expression; the attributes are not checked.

condition	Description
EQ	Equal; the value of first expression must be equal to the value of the second expression. The expressions must both be one of the following: Absolute Immobile relative to the same section Relocatable reative to the same section External relative to the same external symbol The word-address, parcel-address, or value attributes must be the same.
NE	Not equal; the first expression and the second expression do not satisfy the conditions required for EQ described above.

count Statement count; must be an absolute expression with positive value. All symbols in the expression, if any, must be previously defined. A missing or null count subfield gives a zero count.

count is only used when the location field is not specified. If *name* is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

Example:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	TEST	
SYM1	=	0	
SYM2	=	*	
SYM3	SET	1000	
SYM4	SET	500	

Example (continued):

```
|Location|Result    |Operand        |Comment
|1       |10        |20             |35
|        |          |               |
|NOTEQ   |IFE       |SYM1,EQ,SYM2   |; Condition fails, values are
|        |S1        |SYM1           |; the same, but the attributes
|        |S2        |SYM2           |; are different
|NOTEQ   |ELSE      |               |; The ELSE sequence is assembled
|        |S1        |SYM3           |; Statement included
|        |S2        |SYM4           |; Statement included
|NOTEQ   |ENDIF     |               |; End of conditional sequence
|        |END       |               |
```

5.9.4 IFM - TEST MACHINE CHARACTERISTICS

The IFM pseudo instruction tests characteristics of the current target machine. If the result of the machine condition is true, assembly continues with the next statement. If the result of the machine condition is false, subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

If any errors are encountered while evaluating the string condition, the resulting condition is handled as if true and an appropriate listing message is issued.

The IFM pseudo instruction can be specified anywhere within a program segment. If the IFM pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the IFM pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result    |Operand
|        |          |
|[name]  |IFM       |logical-name["[count]]
|[name]  |IFM       |numeric-name","condition"["[expression]["[count]]
|        |          |
|[name]  |ifm       |logical-name["[count]]
|[name]  |ifm       |numeric-name","condition"["[expression]["[count]]
```

name Optional name of conditional sequence of code. A conditional sequence of code controlled by a *name* is ended by an ENDIF pseudo with a matching name. The condition of a conditional sequence of code controlled by a *name* can be reversed by an ELSE pseudo with a matching name. If both *name* and *count* are present, *name* takes precedence.

name must meet the requirements for names as described in the BNF. For a description, see subsection 4.2, Names.

logical-name

Logical name; the mnemonic signifying a logical condition of the machine for which CAL is currently targeting code. For a detailed list of the mnemonics, refer to the logical traits of the CPU option for the appropriate operating system (section 2, Operating Systems)

numeric-name

Numeric name; a mnemonic signifying a numeric condition of the machine for which CAL is currently targeting code. For a detailed list of the mnemonics, refer to the numeric traits of the CPU option for the appropriate operating system (section 2, Operating Systems). These mnemonics may be specified in mixed case.

condition

Specifies the relation to be satisfied between the numeric name and the expression, if any. *condition* can be entered in mixed case and must be one of the following:

condition	Description
LT	Less than; the value of the numeric name must be less than the expression.
LE	Less than or equal to; the value of the numeric name must be less than or equal to the expression.
GT	Greater than; the value of the numeric name must be greater than the expression.
GE	Greater than or equal to; the value of the numeric name must be greater than or equal to the expression.
EQ	Equal; the value of the numeric name must be equal to the expression.
NE	Not equal; the value of the numeric name must not be equal to the expression.

expression

Expression to be compared to the numeric name. All symbols in the expression must be previously defined and must have an address attribute of value and a relative attribute of absolute. If the current base is mixed, a default of decimal is used. If an expression is not specified, the absolute value of 0 is used.

Expressions must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

count Statement count; must be an absolute expression with positive value. All symbols in the expression, if any, must be previously defined. A missing or null count subfield gives a zero count.

count is only used when the location field is not specified. If *name* is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

Example:

```
|Location|Result    |Operand        |Comment
|1       |10        |20             |35
|        |          |               |
|        |ident     |test           |
|ex1     |ifm       |vpop           |; Assuming the condition is
|        |.         |               |; true, skipping does not
|        |.         |               |; occur within the IFM part.
|        |.         |               |
|ex1     |endif     |               |
|ex2     |ifm       |numcpus,eq,4   |; Assuming the condition is
|        |.         |               |; false, skipping occurs.
|        |.         |               |
|        |.         |               |
|ex2     |else      |               |; Toggles the condition so
|        |.         |               |; the else part is not skipped.
|        |.         |               |
|        |.         |               |
|ex2     |endif     |               |
|        |          |               |
|        |end       |               |
```

5.9.5 SKIP - UNCONDITIONALLY SKIP STATEMENTS

The SKIP pseudo instruction unconditionally skips subsequent statements. If a location field name is present, skipping stops when an ENDIF or ELSE with the same name is encountered; otherwise, skipping stops when the statement count is exhausted.

The SKIP pseudo instruction can be specified anywhere within a program segment. If the SKIP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the SKIP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*name*]	SKIP	[*count*]
[*name*]	skip	[*count*]

name Optional name of conditional sequence of code. If both *name* and *count* are present, *name* takes precedence. *name* must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

count Statement count; must be an absolute expression with positive value. All symbols in the expression, if any, must be previously defined. A missing or null count subfield gives a zero count.

count is only used when the location field is not specified. If *name* is not present and *count* is present in the operand field, skipping stops when *count* is exhausted. If neither *name* nor *count* is present, no skipping occurs.

Example:

Location	Result	Operand	Comment
1	10	20	35
	SKIP		; No skipping occurs
SNAME1	SKIP		; Statements are skipped until
	.	.	; an ENDIF or ELSE with a
	.	.	; location field label that
	.	.	; matches SNAME1 is found.

Example (continued):

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|SNAME1  |ENDIF     |           |
|        |.         |.          |
|        |.         |.          |
|        |.         |.          |
|        |          |           |
|SNAME2  |SKIP      |10         |; Statements are skipped until
|        |.         |.          |; an ENDIF or ELSE with a
|        |.         |.          |; location field label that
|        |.         |.          |; matches SNAME2 is found.
|SNAME2  |ENDIF     |           |
|        |.         |.          |
|        |.         |.          |
|        |.         |.          |
|        |SKIP      |4          |; Four statements are skipped.
```

5.9.6 ENDIF - END CONDITIONAL CODE SEQUENCE

The ENDIF pseudo instruction terminates skipping initiated by an IFA, IFE, IFC, ELSE, or SKIP pseudo instruction with the same location field name. Otherwise, ENDIF acts as a do-nothing pseudo instruction. ENDIF has no effect on skipping, which is controlled by a statement count.

The ENDIF pseudo instruction can be specified anywhere within a program segment. Skipping is terminated by an ENDIF pseudo instruction with a matching location field name. If the ENDIF pseudo is found within a definition, it is defined and is not recognized as a pseudo instruction.

Format:

```
|Location|Result    |Operand
|        |          |
|name    |ENDIF     |ignored
|name    |endif     |ignored
```

name Required name of conditional sequence of code. *name* must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

NOTE

If an END pseudo instruction is encountered in a skipping sequence, an error message is issued and skipping is continued. An END should not be used within a skipping sequence.

5.9.7 ELSE - TOGGLE ASSEMBLY CONDITION

The ELSE pseudo instruction terminates skipping initiated by an IFA, IFC, IFE, ELSE, or SKIP pseudo instruction with the same location field name. If statements are currently being skipped under control of a statement count, ELSE has no effect.

The ELSE pseudo instruction can be specified anywhere within a program segment. If the assembler is not currently skipping statements, ELSE initiates skipping. Skipping is terminated by an ELSE pseudo instruction with a matching location field name. If the ELSE pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
name	ELSE	ignored
name	else	ignored

name Required name of conditional sequence of code. *name* must meet the requirements given for names as described in the BNF. For a description of names, see subsection 4.2.

Examples:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|SYM     |=         |1                |
|L       |MICRO     |'LESS THAN'      |
|DEF     |=         |1000             |
|BUF     |=         |100              |
|        |IFA       |#DEF,A,1         |
|A       |=         |10               |
|BTEST   |IFA       |EXT,SYM          |
|WARNING |ERROR     |                 |; Generate warning message if
|        |          |                 |; SYM is absolute
|BTEST   |ELSE      |                 |
|        |A1        |SYM              |; Assemble if SYM not absolute
|        |          |                 |
|BTEST   |ENDIF     |                 |
|        |          |                 |
|* Assemble BSSZ instruction if W.* is less than BUF, otherwise
|* assemble ORG     |                 |
|        |IFE       |W.*,LT,BUF,2     |
|        |BSSZ      |BUF-W.*          |; Generate words of zero to
|*       |          |                 |; address BUF
|        |SKIP      |1                |; Skip next statement
|        |ORG       |BUF              |
|        |IFC       |'"L"',EQ,,2      |
|ERROR   |ERROR     |                 |; Error message; if micro string
|*       |          |                 |; defined by L is empty
|X       |IFC       |'ABCD',GT,'ABC'  |; If ABCD is greater than ABC
|        |S1        |DEF              |; Statement is included
|        |S2        |BUF              |; Statement is included
|X       |ENDIF     |                 |
|Y       |IFC       |' ',GT,,2        |; If single space is greater
|*       |          |                 |; than null string.
|        |S3        |DEF              |; Statement is included
|        |S4        |BUF              |; Statement is included
|Z       |IFC       |'''',EQ,*'*,2    |; If single apostrophe equals
|*       |          |                 |; single apostrophe
|        |S5        |5                |
|        |S6        |6                |
```

5.10 MICROS

Through the use of micros, a programmer is able to assign a name to a character string and subsequently refer to the character string through use of its name. A reference to a micro results in the character string

being substituted for the name before assembly of the source statement containing the reference.

- CMICRO Constant micro; assigns a name to a character string
- MICRO Redefinable micro; assigns a name to a character string
- OCTMIC Converts the octal value of an expression to a character string, and assigns it a redefinable name.
- DECMIC Converts the decimal value of an expression to a character string, and assigns it a redefinable micro name.

In addition to the micros previously listed, the CAL assembler provides the following predefined micros.

$DATE Current date - '*mm/dd/yy*'

$JDATE Julian date - '*yyddd*'

$TIME Time of day - '*hh:mm:ss*'

$MIC Micro character - double quote mark (")

$CNC Concatenation character - underscore (_)

$QUAL Name of qualifier in effect; if none, then null string.

$CPU Target machine: 'CRAY 1', 'CRAY XMP', or 'CRAY 2'

$CMNT Comment character used with the new format - semicolon (;)

$APP Append character used with the new format - circumflex (^)

Example:

The following example illustrates the use of a predefined micro, $DATE.

```
|Location|Result    |Operand              |Comment
|1       |10        |20                   |35
|        |          |                     |
|        |DATA      |'THE DATE IS "$DATE"'
|        |DATA      |'THE DATE IS 06/23/82'†
```

† Generated by CAL

Micros can be referenced anywhere in a source statement, except a comment, by enclosing the micro name in quotation marks. If column 72 of a line is exceeded as a result of a micro substitution, the assembler creates additional continuation lines. No replacement takes place if the micro name is unknown or if one of the micro marks has been omitted.

Example:

A micro named PFX is defined as the character string ID. A reference to PFX is in the location field of a line:

```
|Location|Result    |Operand         |Comment
|1       |10        |20              |35
|        |          |                |
|"PFX"TAG|S0        |S1              |; Left-shifted three spaces
|        |          |                |; when edited
```

However, before the line is interpreted, CAL substitutes the definition for PFX producing the following line:

```
|Location|Result    |Operand         |Comment
|1       |10        |20              |35
|        |          |                |
|IDTAG S0          S1               ; Left-shifted three spaces
|        |          |                |; when edited
```

5.10.1 CMICRO - CONSTANT MICRO DEFINITION

The CMICRO pseudo assigns a name to a character string. Once the name is defined, it cannot be redefined.

If the CMICRO pseudo instruction is defined within the global definitions part of a program segment, it can be referenced at any time after its definition by any of the segments that follow. If the CMICRO pseudo instruction is defined within a program module, it can be referenced at any time after its definition within the module. However, a constant micro defined within a program module is discarded at the end of the module and cannot be referenced by any segments that follow.

If the CMICRO pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the CMICRO pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
name	CMICRO	[*string*["[,"[*exp*]["[,"[*exp*]["[,"[*case*]]]]]
name	cmicro	[*string*["[,"[*exp*]["[,"[*exp*]["[,"[*case*]]]]]

name Required micro name. *name* is assigned to the character string found in the operand field and has nonredefinable attributes. If *name* has been previously defined and the string represented by the previous definition is not the same string, an error message is issued and definition occurs. If the strings match, no error message is issued and no definition occurs.

name must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

string Optional character string, which can include previously defined micros. If *string* is not specified, an empty string is used.

A character string can be delimited by any character other than a space. Two consecutive occurrences of the delimiting character indicate a single such character. For example, a micro consisting of the single character * could be specified as '*' or ****.

exp Optional expressions. The first expression must be an absolute expression indicating the number of characters in the micro character string. All symbols, if any, must be previously defined. If the current base is mixed, decimal is used for the expression.

The micro character string is terminated by the value of the first expression or the final apostrophe of the character string, whichever occurs first. The string is considered empty if the first expression has a 0 or negative value. If the first expression is not specified, the full value of the character string is used. In this case the string is terminated by the final apostrophe.

The second expression must be an absolute expression indicating the micro string's starting character. All symbols, if any, must be previously defined. If the current base is mixed, decimal is used for the expression.

The starting character of the micro string begins with the character that is equal to the value of the second expression, or with the first character in the character string if the second expression is null or has a value of 1 or less.

The expressions must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

case Optional character conversion case. *case* denotes how uppercase and lowercase characters are interpreted when they are read from *string*. Character conversion is restricted to the letter characters (A-Z and a-z) specified in *string*. *case* can be specified in uppercase, lowercase, or mixed case and must be one of the following:

MIXED Default; No uppercase or lowercase conversions are made; *string* is interpreted as entered.

UPPER If UPPER is specified, all lowercase letter characters in *string* are converted to their uppercase equivalents.

LOWER If LOWER is specified, all uppercase letter characters in *string* are converted to their lowercase equivalents.

5.10.2 MICRO - MICRO DEFINITION

The MICRO pseudo instruction assigns a name to a character string. The assigned name can be redefined.

A redefinable micro can be referenced and redefined after its intitial definition within a program segment. A micro defined with the MICRO pseudo instruction is discarded at the end of a module and cannot be referenced by any of the following segments.

The MICRO pseudo instruction can be specified anywhere within a program segment. If the MICRO pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the MICRO pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Formats:

Location	Result	Operand
name	MICRO	[*string*[","[*exp*][","[*exp*][","[*case*]]]]]
name	micro	[*string*[","[*exp*][","[*exp*][","[*case*]]]]]

name Required micro name. *name* is assigned to the character string found in the operand field and has redefinable attributes. If *name* has been previously defined, the previous micro definition is lost.

string Optional character string, which can include previously defined micros. If *string* is not present, an empty string is used.

A character string can be delimited by any character other than a space. Two consecutive occurrences of the delimiting character indicate a single such character. For example, a micro consisting of the single character * could be specified as '*' or ****.

exp Optional expressions. The first expression must be an absolute expression indicating the number of characters in the micro character string. All symbols, if any, must be previously defined. If the current base is mixed, decimal is used for the expression.

The micro character string is terminated by the value of the first expression or the final apostrophe of the character string, whichever occurs first. The string is considered empty if the first expression has a 0 or negative value. If the first expression is not specified, the full value of the character string is used. In this case the string is terminated by the final apostrophe.

The second expression must be an absolute expression indicating the micro string's starting character. All symbols, if any, must be previously defined. If the current base is mixed, decimal is used for the expression.

The starting character of the micro string begins with the character that is equal to the value of the second expression, or with the first character in the character string if the second expression is null or has a value of 1 or less.

The expressions must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

case Optional character conversion case. *case* denotes how uppercase and lowercase characters are interpreted when they are read from *string*. Character conversion is restricted to the letter characters (A-Z and a-z) specified in *string*. *case* can be specified in uppercase, lowercase, or mixed case and must be one of the following:

MIXED Default; No uppercase or lowercase conversions are made; *string* is interpreted as entered.

UPPER If UPPER is specified, all lowercase letter characters in *string* are converted to their uppercase equivalents.

LOWER If LOWER is specified, all uppercase letter characters in *string* are converted to their lowercase equivalents.

Example:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|MIC     |MICRO     |'THIS IS A MICRO STRING'
|MIC2    |MICRO     |'"MIC"',1        |
|MIC2†   |MICRO     |'THIS IS A MICRO STRING',1
|MIC3    |MICRO     |'"MIC2"'         |
|MIC3†   |MICRO     |'T'              |
|MIC4    |MICRO     |'"MIC"',10       |; Call to micro MIC2
|MIC4†   |MICRO     |'THIS IS A MICRO STRING',10
|MIC5    |MICRO     |'"MIC4"'         |
|MIC5†   |MICRO     |'THIS IS A '     |
|MIC6    |MICRO     |'"MIC"',5,11     |
|MIC6†   |MICRO     |'THIS IS A MICRO STRING',5,11
|MIC7    |MICRO     |'"MIC6"'         |
|MIC7†   |MICRO     |'MICRO'          |
|MIC8    |MICRO     |'"MIC"',11,5     |
|MIC8†   |MICRO     |'THIS IS A MICRO STRING',11,5
|MIC9    |MICRO     |'"MIC8"'         |
|MIC9†   |MICRO     |' IS A MICRO'    |
```

† These lines have been edited by CAL.

5.10.3 OCTMIC - OCTAL MICROS

The OCTMIC pseudo instruction converts a value of an expression into a character string that is assigned a redefinable micro name. The character string that is generated by the pseudo instruction is represented as an octal number. The final length of the micro string is inserted into the code field of the listing.

OCTMIC can be specified with zero, one, or two expressions. The value of the first expression is converted to a micro string with a character length equal to the second expression. If the second expression is not specified, the minimum number of characters needed to represent the octal value of the first expression is used.

If the second expression is specified, the string is equal to the length specified by the second expression. If the number of characters in the micro string is less than the value of the second expression, the character value is right justified with the specified fill characters (zeros or blanks) preceding the value. If the number of characters in the string is greater than the value of the second expression, the beginning characters of the string are truncated and a warning message is issued.

The OCTMIC pseudo instruction can be specified anywhere within a program segment. If the OCTMIC pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the OCTMIC pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
name	OCTMIC	[*expression*$_1$]["[,"[*expression*$_2$["[,"[*option*]]]]
name	octmic	[*expression*$_1$]["[,"[*expression*$_2$["[,"[*option*]]]]

name Micro name. *name* must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

expression$_1$
Optional expression; micro string equal to the value of the expression. If specified, *expression*$_1$ must have an address attribute of value and a relative attribute of absolute with all symbols, if any, previously defined. If the current base is mixed, a default of octal is used. If the first expression is not specified the absolute value of zero is used in the creation of the micro string.

expression$_1$ must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

expression$_2$
Optional expression. *expression*$_2$ provides a positive character count less than or equal to decimal 22. If this parameter is present, leading zeros are zeros or blanks (depending on *option*) are supplied, if necessary, to provide the requested number of characters.

If specified, *expression*$_2$ must have an address attribute of value and a relative attriube of absolute with all symbols, if any, previously defined. If the current base is mixed, a default of decimal is used. If the *expression*$_2$ is not specified, the micro string is represented in the minimum number of characters needed to represent the octal value of the first expression.

expression$_2$ must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

option If the second expression is present and fill is needed, *option* represents the type of fill characters (ZERO for zeros or BLANK for spaces) to be used; the default is ZERO. *option* can be entered in mixed case.

Example:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |IDENT     |EXOCT            |
|        |BASE      |O                |; The base is octal.
|ONE     |OCTMIC    |1,2              |;
|_*      |"ONE"     |                 |; Returns 1 in two digits
|*       01         |            ; Returns 1 in two digists
|TWO     |OCTMIC    |5*7+60+700,3     |
|_*      |"TWO"     |                 |; Returns 1023 in three digits
|*       023        |                 ; Returns 1023 in three digits
|THREE   |OCTMIC    |256000,10,ZERO   |
|_*      |"THREE"   |                 |; Zero fill on the left
|*       00256000   |                 ; Zero fill on the left
|FOUR    |OCTMIC    |256000,10,BLANK  |
|_*      |"FOUR"    |                 |; Blank fill (^) on the left
|*       ^^256000   |                 | ; Blank fill (^) on the left
|        |END       |                 |
```

5.10.4 DECMIC - DECIMAL MICROS

The DECMIC pseudo instruction converts the positive or negative value of an expression into a positive or negative decimal character string that is assigned a redefinable micro name. The final length of the micro string is inserted into the code field of the listing.

DECMIC can be specified with zero, one, or two expressions. DECMIC converts the value of the first expression into a character string with a character length indicated by the second expression. If the second expression is not specified, the minimum number of characters needed to represent the decimal value of the first expression is used.

If the second expression is specified, the string is equal to the length specified by the second expression. If the number of characters in the micro string is less than the value of the second expression, and the value of the first expression is positive, the character value is right justified with the specified fill characters (zeros or blanks) preceding the value.

If the number of characters in the string is less than the value of the second expression, and the value of the first expression is negative, a minus sign precedes the value. If zero fill is indicated, zeros are used as fill between the minus sign and the value. If blank fill is indicated, blanks are used as fill before the minus sign.

If the number of characters in the string is greater than the value of the second expression, the characters at the beginning of the string are truncated and a warning message is issued.

The DECMIC pseudo instruction can be specified anywhere within a program segment. If the DECMIC pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the DECMIC pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
name	DECMIC	[*expression*$_1$]["[,"[*expression*$_2$]["[,"[*option*]]]]]
name	decmic	[*expression*$_1$]["[,"[*expression*$_2$]["[,"[*option*]]]]]

name Required micro name. *name* is assigned to the character string representing the decimal value of the *expression*$_1$ and has redefinable attributes.

name must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

expression$_1$

Optional expression; micro string equal to the value of the expression. If specified, *expression*$_1$ must have an address attribute of value and a relative attribute of absolute with all symbols, if any, previously defined.

If the first expression is not specified, the absolute value of zero is used. If the current base is mixed, a default of octal is used. If the first expression is not specified the absolute value of zero is used in the creation of the micro string.

expression$_1$ must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

expression$_2$

Optional expression. *expression*$_2$ provides a positive character count less than or equal to decimal 20. If this parameter is present, leading zeros are zeros or blanks (depending on *option*) are supplied, if necessary, to provide the requested number of characters.

If specified, *expression*$_2$ must have an address attribute of value and a relative attriube of absolute with all symbols, if any, previously defined. If the current base is mixed, a default of decimal is used.

If *expression*$_2$ is not specified, the micro string is represented in the minimum number of characters needed to represent the decimal value of the first expression.

expression$_2$ must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

option If the second expression is present and fill is needed, *option* represents the type of fill characters (ZERO for zeros or BLANK for spaces) to be used; the default is ZERO. *option* can be entered in mixed case.

Examples:

1. The following example includes the DECMIC and MICSIZE pseudo instructions:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|MIC     |MICRO     |'ABCD'           |
|V       |MICSIZE   |MIC              |; The value of V is the number
|        |          |                 |; of characters in the micro
|        |          |                 |; string represented by MIC
|DECT    |DECMIC    |V,2              |; DECT is a micro name.
|-*  There are "DECT" characters in MIC.
|*  There are 19 characters in MIC.†
```

2. The following example demonstrates the ZERO and BLANK options with positive and negative strings:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |BASE      |D                |; The base is decimal.
|ONE     |DECMIC    |1,2              |
|_*      |"ONE"     |                 |; Returns 1 in two digits
|*       01         |                 ; Returns 1 in two digists
|TWO     |DECMIC    |5*8+60+900,3     |; Decimal 1000
|_*      |"TWO"     |                 |; Returns 1000 as 3 digits (000)
|*       000        |                 ; Returns 1000 as 3 digits (000)
|THREE   |DECMIC    |-256000,10,ZERO; Decimal string with zero fill
|_*      |"THREE"   |                 |; Minus sign, zero fill, value
|*       -000256000 |                 ; Minus sign, zero fill, value
|FOUR    |DECMIC    |-256000,10,BLANK; Decimal string with blank fill
|_*      |"FOUR"    |                 |; Blank fill, minus sign, value
|*       ^^^-256000 |                 |   ; Blank fill, minus sign, value
|FIVE    |DECMIC    |256000,10,ZERO   |
|_*      |"FIVE"    |                 | ; Zero fill on the left
|*       0000256000 |                 |    ; Zero fill on the left
|SIX     |DECMIC    |256000,10,BLANK|
|_*      |"SIX"     |                 |; Blank fill (^) on the left
|*       ^^^^256000 |                 |    ; Blank fill (^) on the left
|        |END       |                 |
|SEVEN   |DECMIC    |256000,5         |
|_*      |"SEVEN"   |                 | ; Truncation warning issued
|*       56000      |               ; Truncation warning issued
|EIGHT   |DECMIC    |777777777,3      |
|_*      |"EIGHT"   |                 |; Truncation warning issued
|*       777        |           ; Truncation warning issued
```

† Generated by CAL

5.11 FILE CONTROL (INCLUDE PSEUDO)

The INCLUDE pseudo inserts a file at the current source position. The INCLUDE pseudo always prepares the file for reading by opening it and positioning the pointer at its beginning.

Files can be included within other files. The same file can be included more than once, although CAL does not allow a file that is currently being included to be included again until it has been fully included.

The INCLUDE pseudo instruction can be specified anywhere within a program segment. If the INCLUDE pseudo occurs within a definition, it is recognized as a pseudo instruction and the named file is included in the definition. If the INCLUDE pseudo instruction occurs within a skipping sequence, it is recognized as a pseudo instruction and the named file is included in the skipping sequence. The INCLUDE pseudo statement itself is not inserted into a defined sequence of code.

NOTE

The INCLUDE pseudo can be forced into a definition or skipped sequence of code although it is not recommended for a definition. To do this, embed an underscore (_) anywhere within the pseudo as shown in the following example:

```
IN_CLUDE
```

If editing is enabled during expansion, the files specified with the INLCUDE are included during expansion. However, formal parameters are not substituted into statements when these files are included at expansion time.

Format:

Location	Result	Operand
ignored	INCLUDE	*string*
ignored	include	*string*

string An ASCII character string that identifies the file to be included. The ASCII character string must be a valid file name depending on the operating system under which CAL is executing. If the ASCII character string is not a valid file name or CAL is not able to open the file, a listing message is issued.

string must be specified with a single matching character on both ends. *string* can be delimited by any ASCII character other than a comma or space. Two consecutive occurrences of the delimiting character indicate a single such character is to be included in the character string.

Examples:

1. In the following example, the module named INCTEST contains an INCLUDE pseudo instruction. The file to be included is called DOG and includes the file CAT.

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |IDENT     |INCTEST     |
|        |INCLUDE   |*DOG*       |; Call file DOG with INCLUDE
|        |END       |            |
```

File DOG:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |S1        |1           |; Register S1 gets 1.
|        |INCLUDE   |'CAT'       |; Call file CAT with INCLUDE
|        |S2        |2           |; Register S2 gets 2.
```

File CAT:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |S3        |3           |; Register S3 gets 3.
```

Expansion:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|        |IDENT     |INCTEST    |
|        |INCLUDE   |*DOG*      |; Call file DOG with INCLUDE
|        |S1        |1          |; Register S1 gets 1.
|        |INCLUDE   |'CAT'      |; Call file CAT with INCLUDE
|        |S3        |3          |; Register S3 gets 3.
|        |S2        |2          |; Register S2 gets 2.
|        |END       |           |
```

2. The following example demonstrates that it is illegal to include a file from a file that was included by that file.

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|        |IDENT     |INCTEST    |
|        |INCLUDE   |#DOG#      |; Call file DOG with INCLUDE
|        |END       |           |
```

File DOG:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|        |S1        |1          |; Register S1 gets 1.
|        |INCLUDE   |*CAT*      |; Call file CAT with INCLUDE
|        |S2        |2          |; Register S2 gets 2.
```

File CAT:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|        |S3        |3          |
|        |INCLUDE   |-DOG-      |; Illegal; If file B was
|        |          |           |; included by file A, it cannot
|        |          |           |; include file A.
```

3. The following example demonstrates that it is legal to include a file more than once as long as it is not currently being included:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |ident     |inctest     |
|        |include   |%dog%       |; Call file dog with include
|        |include   |%dog%       |; Call file dog with include
|        |end       |            |
```

File DOG:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |s1        |1           |; Register S1 gets 1.
|        |s2        |2           |; Register S2 gets 2.
```

Expansion:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |ident     |inctest     |
|        |include   |dog         |; Call file dog with include
|        |s1        |1           |; Register S1 gets 1.
|        |s2        |2           |; Register S2 gets 2.
|        |include   |dog         |; Call file dog with include
|        |s1        |1           |; Register S1 gets 1.
|        |s2        |2           |; Register S2 gets 2.
|        |end       |            |
```

5.12 DEFINED SEQUENCES

You can define sequences of instructions to be saved for assembly at a later point in the source program. The four types of defined sequences are as follows: MACRO, OPDEF, DUP, and ECHO. Defined sequences have several functional similarities.

Since the ENDM, ENDDUP, and STOPDUP pseudo instructions terminate defined sequences and the LOCAL and OPSYN pseudo instructions are associated with definitions, they are included in this subsection. The following is a brief description of the pseudo instructions included in this subsection.

- MACRO A sequence of source program instructions that are saved by the assembler for inclusion in a program when called for by the macro name. The macro call resembles a pseudo instruction.

- OPDEF A sequence of source program instructions that are saved by the assembler for inclusion in a program called for by the OPDEF instruction. The opdef resembles a symbolic machine instruction.

- DUP Introduces a sequence of code that is assembled repetitively for a specific count; the duplicated code immediately follows the DUP pseudo instruction.

- ECHO Introduces a sequence of code that is assembled repetitively until an argument list is exhausted

- ENDM Ends a macro or opdef definition

- ENDDUP Terminates a dup or echo sequence of code

- STOPDUP Stops the duplication of a code sequence by overriding the repetition condition

- LOCAL Specifies unique strings that are usually used as symbols within a macro, opdef, dup, or echo

- OPSYN Defines a location field functional that is the same as a named operation in the operand field functional

5.12.1 SIMILARITIES AMONG DEFINED SEQUENCES

Defined sequences have the following functional similarities:

- Editing
- Definition format
- Formal parameters
- Instruction calls
- Interact with the INCLUDE pseudo instruction

5.12.1.1 Editing

Editing is disabled by the assembler at definition time. The body of the definition (see definition format) is saved before micros and concatenation marks are edited. Editing occurs when the definition is assembled each time it is called if editing is enabled. The ENDDUP, ENDM, END, INCLUDE, and LOCAL pseudo instructions and prototype statements should not contain any micros or concatenation characters, because they may not be recognized at definition time.

When a sequence is defined, editing is disabled and cannot be explicitly enabled. When a sequence is called, CAL performs the following operations:

- Checks for all parameter substitutions that were marked at definition time
- Edits the statement if editing is enabled
- Processes the statement

It is possible to take advantage of the fact that editing is disabled at definition time. If, for example, the INCLUDE pseudo instruction is specified with embedded blanks as shown in macro INC, a saving in program overhead is achieved.

Example:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |MACRO     |            |
|        |INC       |            |
|        |.         |.           |
|        |.         |.           |
|        |.         |.           |
|        |IN_CLUDE  |MYFILE      |; INCLUDE pseudo with an
|        |.         |.           |; embedded underscore
|        |.         |.           |
|        |.         |.           |
|INC     |ENDM      |            |
```

Since editing is disabled at definition time, concatenation does not occur until INC is called. If editing is enabled when the macro is called, MYFILE is included at that time.

Embedding blanks in an INCLUDE pseudo becomes desirable when the INCLUDE pseudo identifies large files. Since files are included when the macro is called and not at definition time, embedding blanks in the INCLUDE pseudo instruction can reduce the overhead required for a program.

5.12.1.2 Definition format

Macro, opdef, dup, and echo use the same definition format. The format consists of a header, body, and end.

The header consists of a MACRO, OPDEF, DUP, or ECHO pseudo instruction, a prototype statement for a macro or opdef definition, and, optionally, LOCAL pseudo instructions. For a macro, the prototype statement provides a functional and a list of formal parameters. For an opdef, the prototype statement supplies the syntax and the formal parameters.

LOCAL pseudo instructions identify parameter names that CAL must make unique to the assembly each time the definition sequence is placed in a program segment. Asterisk comments can be placed in the header and have no affect on the way CAL scans the header. Asterisk comments are dropped from the definition. To force asterisk comments into a definition, see subsection 3.3.5, Comment.

The body of the definition begins with the first statement following the header. The body can consist of a series of CAL instructions other than an END pseudo. The body of a definition can be empty, or it may include other definitions and calls. However, a definition used within another definition is not recognized until the definition in which it is contained is called. Therefore, an inner definition cannot be called before the outer definition is called for the first time.

A comment statement identified by an asterisk in column 1 is ignored in the definition header and the definition body. Such comments are not saved as a part of the definition sequence. Comment fields on other statements in the body of a definition are saved.

An ENDM pseudo instruction with the proper name in the location field ends a macro or opdef definition. A statement count or an ENDDUP pseudo instruction with the proper name in the location field ends a dup definition. An ENDDUP pseudo instruction with the proper name in the location field ends an echo definition.

5.12.1.3 Formal parameters

Formal parameters are defined in the definition header and recognized in the definition body. Four types of formal parameters are recognized: positional, keyword, echo, and local.

The characters that identify positional, keyword, echo, and local parameters must all have unique names within a given definition. In addition, positional, keyword, and echo parameters are case sensitive. These parameters must be specified in the body of the definition exactly as specified in the definition header to be recognized.

Parameter names must meet the requirements for identifiers as described in the BNF. For a description of names, see subsection 4.2.

A formal parameter name can be embedded within the definition body. However, embedded parameters must safisfy the following requirements:

- The first character of an embedded parameter must begin with a legal initial-identifier-character.

- An embedded parameter cannot be preceded by an initial-identifier-character. For example, PARAM is a legally embedded parameter within the string ABC_PARAM_DEF, because it is preceded by an underscore character. PARAM is not a legally embedded character within the string ABCPARAMDEF, because it is preceded by an initial-identifier-character (C).

- An embedded parameter must not be followed by an identifier-character. In the following example, the embedded parameter is legal, because it is followed by a element separator (blank character):

  PARAM678

  In the following example, the embedded parameter is illegal, because it is followed by an identifier-character:

  PARAM6789

  Embedded parameters must contain eight or fewer characters. PARAM6789 is illegal, because it is nine characters long. The character that follows an embedded parameter (9) cannot be an identifier-character.

- An embedded parameter must occur before the first comment character (;) of each statement within the body, if and only if the new format is specified.

- An embedded parameter must have a matching formal parameter name in the definition header.

Formal parameter names should not be END, ENDM, ENDDUP, LOCAL, or INCLUDE. If any of these are used as parameter names, substitution of actual arguments occur when these names are contained in any inner definition when the definition is referenced.

NOTE

Arguments are not substituted for formal parameters into statements within included files if the file is included at expansion time.

5.12.1.4 Instruction calls

Each time a definition sequence of code is called, an entry is added to a list of currently active defined sequences within the assembler. The most recent entry indicates the current source of statements to be assembled. When a definition is called within a definition sequence being assembled, another entry addition is made to the list of defined sequences, and assembly continues with the new definition sequence belonging to the inner, or nested, call.

When the end of a definition sequence is reached, the most recent list entry is removed and assembly continues with the previous list entry. When the list of defined sequences is exhausted, assembly continues with statements from the source file.

An inner nested call can be recursive; that is, it can reference the same definition that is referenced by an outer call. The depth of nested calls permitted by CAL is limited only by the amount of memory available.

The sequence field in the right margin of the listing shows the definition name and nesting depth for defined sequences being assembled. Nesting depth numbers begin in column 89 and can be one of the following: :1, :2, :3, :4, :5, :6, :7, :8, :9, :*.

If the nesting depth is greater than 9, CAL keeps track of the current nesting level and an asterisk represents nesting depths of 10 or more. Nesting depth numbers are restricted to two characters so that only the two right-most character positions are overwritten.

If the sequence field (columns 73 through 90) is not empty in the source file, CAL copies the existing field for a call into every statement expanded by the call with columns 89 and 90 reserved for the nesting level. For example, if the sequence field for MCALL was LQ5992A.112, the sequence field for a statement expanded from MCALL would read as follows:

```
    LQ5992A.112     :1
```

Additional nested calls within MCALL would change the nesting level, but the sequence field would be unchanged during the MCALL. For example:

```
    LQ5992A.112     :2
    LQ5992A.112     :2
    LQ5992A.112     :2
    LQ5992A.112     :3
    .
    .
    .
    LQ5992A.112     :*
    .
    .
    .
    LQ5992A.112     :1
```

If the sequence field (columns 73 through 90) is empty in the source file, CAL inserts the name of the definition as follows:

- Macro The inserted name in the sequence field is the functional found in the result field of the macro prototype statement.

- Opdef The inserted name in the sequence field is the name used in the location field of the OPDEF pseudo instruction itself.

- Dup The inserted name in the sequence field is the name used in the location field of the DUP pseudo, or if the count is specified and name is not, the name is *Dup.

- Echo The inserted name in the sequence field is the name used in the location field of the ECHO pseudo instruction.

In all cases, the first two columns of the sequence field contain ** to indicate that CAL has generated the sequence field. Columns 89 and 90 of the sequence field are reserved for the nesting level. If, for example, the sequence field is missing for MCALL, it would read as follows:

```
** MCALL        :1
```

Additional nested calls within MCALL would change the nesting level, but the sequence field number would be unchanged for the duration of MCALL.

Example:

```
** MCALL        :1
** MCALL        :2
** MCALL        :2
** MCALL        :2
** MCALL        :3
.
.
.
** MCALL        :*
.
.
.
** MCALL        :1
```

5.12.1.5 INCLUDE pseudo instruction

The INCLUDE pseudo instruction operates with defined sequences as follows:

MACRO INCLUDE pseudo instructions are expanded at definition time.

OPDEF INCLUDE pseudo instructions are expanded at definition time.

DUP INCLUDE pseudo instructions are expanded at definition time. If count is specified, the INCLUDE pseudo statement itself is not included in the statements being counted.

ECHO INCLUDE pseudo instructions are expanded at definition time.

5.12.2 MACRO

A macro definition identifies a sequence of statements that is defined; saved by the assembler for inclusion elsewhere in a program. A macro is referenced at a later point in the source program by a single instruction, the macro call. Each time the macro call occurs, the definition sequence is placed into the source program.

The MACRO pseudo instruction can be specified anywhere within a program segment. If the MACRO pseudo instruction is found within a definition, it is defined. If the MACRO pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

A macro definition is defined as global if it occurs within the global definitions part of a program segment. Macro definitions are local if they occur within a program module (an IDENT, END sequence). A global definition can be redefined locally, but the global definition is reenabled and the local definition is discarded at the end of the program module. A global definition can be referenced anywhere within the assembler program following the definition.

Example:

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|        |MACRO     |GLOBAL        |; Globally defined
|_* GLOBAL DEFINITION IS USED.     |
|GLOBAL  |ENDM      |              |
|        |GLOBAL    |              |; Call to global definition
|        |LIST      |MAC           |
|* GLOBAL DEFINITION IS USED.      |
|        |IDENT     |TEST          |
|        |GLOBAL    |              |; Call to global definition
|* GLOBAL DEFINTION IS USED.       |
|        |MACRO     |GLOBAL        |; Locally defined
|* Redefinition warning message is issued
|* LOCAL DEFINTION IS USED.        |
```

Example (continued):

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|GLOBAL  |ENDM       |                 |
|        |GLOBAL     |                 |; Call to local definition
|* LOCAL DEFINITION IS USED.           |
|        |END        |                 |; Local definitions are discarded
|        |IDENT      |TEST2            |
|        |GLOBAL     |                 |; Call to global definition
|* GLOBAL DEFINTION IS USED.           |
```

5.12.2.1 Macro definition

The macro definition header consists of the MACRO pseudo instruction, a prototype statement, and optional LOCAL pseudo instructions. The prototype statement provides a name for the macro and a list of formal parameters and default arguments.

A comment statement, identified by an asterisk in column 1, is ignored in the definition header or definition body. Such comments are not saved as a part of the definition sequence. Comment fields on other statements in the body of a definition are saved.

The end of a macro definition is signaled by an ENDM pseudo instruction with a functional name that matches the functional name in the result field of the macro prototype statement. For a description of the ENDM pseudo instruction, see ENDM in this subsection 5.12, Defined Sequences.

Example:

The following macro transfers an integer in an A register to an S register. Then it converts it to a normalized floating-point number.

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|        |macro      |                 |
|        |intconv    |p1,p2            |; P1=A reg, P2=S reg
|        |p2         |+f_p1            |; Transfer with special exp
|        |           |                 |; and sign extension
|        |p2         |+f_p2            |; Normalize the S register
|intconv |endm       |                 |; End of macro definition
```

As with every macro, INTCONV begins with the pseudo instruction MACRO. The second statement is the prototype statement; names the macro and defines the parameters. The next three statements are definition statements. They identify what the macro is supposed to do. The ENDM pseudo instruction ends the macro definition.

The format of the macro definition is as follows:

```
|Location|Result    |Operand          |Comment
|        |          |
|ignored |MACRO     |ignored          |; Definition header
|loc     |funct     |parameters       |; Prototype statement
|        |LOCAL     |[name]{","[name]} ; Optional LOCAL pseudos
|        |.         |                 |
|        |.         |                 |; Definition body
|        |.         |                 |
|funct   |ENDM      |                 |; Definition end
```

```
|Location|Result    |Operand          |Comment
|        |          |
|ignored |macro     |ignored          |; Definition header
|loc     |funct     |parameters       |; Prototype statement
|        |local     |[name]{","[name]} ; Optional LOCAL pseudos
|        |.         |                 |
|        |.         |                 |; Definition body
|        |.         |                 |
|funct   |endm      |                 |; Definition end
```

The format of the macro prototype statement is as follows:

```
|Location|Result    |Operand
|        |          |
|loc     |functional |positional-parameters[","[keyword-parameters]]
|loc     |functional |keyword-parameters
```

positional-parameters and *keyword-parameters* are defined as follows:

```
positional-parameters ::=
   [["!"]["*"]name][","positional-parameters] |
   [["*"]["!"]name][","positional-parameters] .
```

```
keyword-parameters ::=
   "!"["*"]name"="[expression-argument-value]
   [","[keyword-parameters]] |
   ["*"]"!"name"="[expression-argument-value]
   [","[keyword-parameters]] |
   ["*"]name"="[string-argument-value]
   [","[keyword-parameters]] .

expression-argument-value ::= expression .

string-argument-value ::= embedded-argument | argument-character† .
```

loc Optional location field parameter. *loc* must be terminated by a space.

loc must meet the requirements for names given in the BNF. For a description of names, see subsection 4.2.

functional

Name of the macro, must be a valid identifier or the equal sign. If *functional* is the same as a currently defined pseudo instruction or macro, this definition redefines the operation associated with *functional*, and a message is issued.

functional must meet the requirements for functionals as described in the BNF. For a description of functionals, see appendix A, Instruction Syntax.

positional-parameters

Positional-parameters must be specified with valid and unique names. *positional-parameters* must meet the requirements for names given in the BNF. For a description of names, see subsection 4.2.

There can be none, one, or more positional-parameters. Positional-parameters cannot be entered after keyword-parameters. The default argument for a positional-parameter is an empty string.

The positional-parameters defined in the macro definition are case sensitive. Positional-parameters specified in the definition body must identically match positional-parameters defined by the macro prototype statement.

---

† Any ASCII character, except a comma, space, or semicolon (new format only), is permitted.

The ! is optional. If the ! is not included, the positional-parameter can take one of the following forms when the macro is called:

- Embedded-argument
- Character string
- Null string

Embedded-argument; a left parenthesis signals the beginning of an embedded argument and must be terminated by a matching right parenthesis. An embedded argument can contain an argument as described in appendix A, Instruction Syntax. Note that an embedded argument can also contain pairs of matching left and right parentheses. If an asterisk preceeds the positional parameter name, the embedded argument is used in its entirety. If an asterisk does not preceed the positional parameter name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

Character string; any character up to but not including a legal terminator (space or semicolon for new format) or an element separator (comma).

Null string; a null argument.

If the ! is included, the parameter can take one of the following forms when the macro is called:

- Syntactically valid expression
- Null string

Syntactically valid expression; an expression can include a legal terminator (space or semicolon for new format) or an element separator (comma). The syntactically valid expression satisfies the requirements for an expression, but it is used only as an argument and is not evaluated in the macro call itself. See the BNF for a description of the syntax of an expression.

If the syntactically valid expression is an embedded-argument, the following will occur. If an asterisk preceeds the positional parameter name, the embedded argument is used in its entirety. If an asterisk does not preceed the positional parameter name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

Null string; a null argument.

Use of the syntactically valid expression, permits you to enter a string (='','R) of characters that may contain one or more spaces or comma. For example:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |MACRO     |                 |
|        |JUSTIFY   |!PARAM           |; Macro prototype
|        |.         |.                |
|        |.         |.                |
|        |.         |.                |
|JUSTIFY |ENDM      |                 |
|        |JUSTIFY   |','R             |; Macro call
|        |JUSTIFY   |'   'R           |; Macro call
```

When the following macro is called, the positional parameter p1 receives a value of v1 because an asterisk does not preceed the parameter on the prototype statement. The positional parameter p2, however, receives a value of (v2) because an asterisk preceeds the parameter on the prototype statement.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |macro     |                 |
|        |paren     |p1,*p2           |; Macro prototype
|        |.         |.                |
|        |.         |.                |
|        |.         |.                |
|paren   |endm      |                 |
|        |paren     |(v1),(v2)        |; Macro call
```

keyword-parameter

Keyword-parameters must be specified with valid and unique names. Names within *keyword-parmeter* must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

There can be none, one, or more keyword-parameters. Names within keyword-parameters can be entered in any order. Default arguments can be provided for each keyword-parameter at definition time and are used if the keyword is not specified at call time.

The keyword-parameters defined in a macro definition are case sensitive. The keyword-parameters specified in the macro body must match the positional-parameters specified in the macro prototype statement.

The ! is optional. If the ! is not included, the parameter can take one of the following forms when the macro is called:

- Embedded-argument
- Character string
- Null string

Embedded-argument; a left parenthesis signals the beginning of an embedded argument and must be terminated by a matching right parenthesis. An embedded argument can contain an argument as described in appendix A, Instruction Syntax.Note that an embedded argument can also contain pairs of matching left and right parentheses. If an asterisk preceeds the positional parameter name, the embedded argument is used in its entirety. If an asterisk does not preceed the positional parameter name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

Character string; any character up to but not including a legal terminator (space or semicolon for new format) or an element separator.

Null string; a null argument.

If the ! is included, the parameter can take one of the following forms when the macro is called:

- Syntactically valid expression
- Null string

Syntactically valid expression; an expression can include a legal terminator (space or semicolon for new format) or an element separator (comma). The syntactically valid expression satisfies the requirements for an expression, but it is used only as an argument and is not evaluated in the macro call itself. See the BNF for a description of the syntax of an expression.

If the syntactically valid expression is an embedded-argument, the following will occur. If an asterisk preceeds the positional parameter name, the embedded argument is used in its entirety. If an asterisk does not preceed the positional parameter name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

Null string; a null argument.

If a default is provided for a keyword-parameter, it must meet the requirements as described above.

5.12.2.2 Macro calls

A macro definition can be called by an instruction of the following format:

Location	Result	Operand
[*locarg*]	*functional*	*positional-arguments* [","[*keyword-arguments*]]
[*locarg*]	*functional*	*keyword-arguments*

positional-arguments and *keyword-arguments* are defined as follows:

positional-argument ::= [*argument*]["," [*positional-arguments*]] .

keyword-arguments ::= *name*"="[*argument*]["," [*keyword-arguments*]]

locarg Optional location field argument. *locarg* must be terminated by a space. *locarg* can be any character up to but not including a space.

If a location field parameter is specified on the macro definition. A matching location field parameter can be specified on the macro call. *locarg* is substituted wherever the location field parameter occurs in the definition. If no location field parameter is specified in the definition, this field must be empty.

functional

Macro functional name; an identifier or an equal sign. *functional* must match the functional specified in the macro definition.

functional must meet the requirements for functionals as described in the BNF. For a description of the BNF for functionals, see appendix A, Instruction Syntax.

positional-arguments

Positional-arguments; an actual argument string corresponding to a positional-parameter that is specified in the definition prototype statement. The requirements for positional-arguments are specified by the corresponding positional-parameter in the macro definition prototype statement. Positional-arguments are not case sensitive to positional-parameters on the macro call.

The first positional-argument is substituted for the first positional-parameter in the prototype operand field, the second positional-argument string is substituted for the second positional-parameter in the prototype operand field, and so on. If the number of positional-arguments is less than the number of positional-parameters in the prototype operand field, null argument strings are used for the missing positional-arguments.

Two consecutive commas indicate a null (empty) positional-argument string.

keyword-arguments
Keyword-arguments; an actual argument string corresponding to a keyword-parameter that is specified in the macro definition prototype statement. The requirements for keyword-arguments are specified by the corresponding keyword-parameter in the macro definition prototype statement.

Keyword-arguments are not recognized until after *n* subfields (*n* commas), where *n* is the number of positional parameters in the operand field of the macro definition.

Keyword-arguments can be listed in any order; matching the order in which keyword-parameters are listed on the macro prototype statement is unnecessary. However, the keyword-parameter is case sensitive and must be specified in the macro call exactly as specified in the macro prototype statement to be recognized.

The default keyword-parameters specified in the macro prototype statement are used as the actual keyword-arguments for missing keyword-arguments.

argument All arguments must meet the requirements of the corresponding parameters as specified in the macro definition prototype statement.

The following restrictions are placed on the macro call:

- The ! is not permitted on the macro call statement. An ! specified in the prototype statement for positional-parameters or keyword-parameters is remembered by CAL when the macro is called.

- If the first character of the actual argument is a left parenthesis, the string must be terminated by a matching right parenthesis. Such an argument is called an embedded-argument and consists of all characters between the enclosing parentheses. An embedded-argument can contain commas and blanks, and can also contain pairs of matching left and right parentheses.

  The actual argument string for each positional and keyword parameter is substituted in the definition sequence wherever the formal parameter occurs. Embedded argument strings are substituted without the enclosing parentheses.

To call a macro, use its name in a code sequence. INTCONV can be called as follows.

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|        |MACRO     |           |
|        |INTCONV   |P1,P2      |; P1=A reg, P2=S reg
|        |P2        |+F_P1      |; Transfer with special expression
|        |          |           |; and sign extension
|        |P2        |+F_P2      |; Normalize the S register
|INTCONV |ENDM      |           |; End of macro definition
|        |LIST      |MAC        |
```

Call and expansion:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|        |INTCONV   |A1,S3      |; Macro call
|        |S2        |+FA1       ; Transfer with special expression
|        |          |           |; and sign extension
|        |S2        |+FS2       ; Normalize the S register
```

NOTE

Comments preceded by an underscore and an asterisk are included in the definition bodies of the following macro examples. These comments are included to illustrate the way in which parameters are passed from the macro call to the macro definition. Since comments are not assembled, _* comments allow arguments to be shown without regard to hardware differences or available machine instructions.

Macro examples:

1. The following examples illustrate the use of positional-parameters and keyword-parameters.

   a. Macro table contains positional and keyword parameters.

```
|Location|Result    |Operand                |Comment
|1       |10        |20                     |35
|        |          |                       |
|        |macro     |                       |
|        |table     |tabn,val1=#0,val2=,val3=0
|tables  |section   |data                   |
|tabn    |con       |'tabn'1                |
|        |con       |val1                   |
|        |con       |val2                   |
|        |con       |val3                   |
|        |section   |*                      |; Resume use of previous section
|table   |endm      |                       |
|        |list      |mac                    |
```

   Call and expansion:

```
|Location|Result    |Operand                |Comment
|1       |10        |20                     |35
|        |          |                       |
|        |table     |taba,val3=4,val2=a ; Macro call
|tables  |section   |data                   |
|taba    |con       |'taba'1                |
|        |con       |#0                     |
|        |con       |a                      |
|        |con       |4                      |
|        |section   |*                      |; Resume use of previous section
```

   b. Macro noorder demonstrates that keyword-parameters are not order dependent.

```
|Location|Result    |Operand                |Comment
|1       |10        |20                     |35
|        |          |                       |
|        |macro     |                       |
|        |noorder   |param1,param2,param3=,param4=b
|        |s1        |param1                 |
|        |s2        |param2                 |
|        |s3        |param3                 |
|        |s4        |param4                 |
|        |list      |mac                    |
```

Call and expansion:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |noorder   |(1),2,param4=dog,param3=d
|        |s1        |1                |
|        |s2        |2                |
|        |s3        |d                |
|        |s4        |dog              |
```

2. Macros ONE, two, and THREE demonstrate that the number of parameters specified in the macro call may be different than the number of parameters specified in the macro definition.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |MACRO     |                 |
|        |ONE       |PARAM1,PARAM2,PARAM3
|_* PARAMETER 1:    |PARAM1           |; SYM1 corresponds to PARAM1
|_* PARAMETER 2:    |PARAM2           |; Null string
|_* PARAMETER 3:    |PARAM3           |; Null string
|ONE     |ENDM      |                 |
|        |LIST      |MAC              |
```

Call and expansion:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |ONE       |SYM1             |; Call using one parameter
|* PARAMETER 1:     |SYM1             ; SYM1 corresponds to PARAM1
|* PARAMETER 2:     |                 ; Null string
|* PARAMETER 3:     |                 ; Null string
```

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |two       |param1,param2,param3
|_* Parameter 1:    |param1           ; Sym1 corresponds to param1
|_* Parameter 2:    |param2           ; Sym2 corresponds to param2
|_* Parameter 3:    |param3           ; Null string
|two     |endm      |                 |
|        |list      |mac              |
```

Call and expansion:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |two       |sym1,sym2        |; Call using two parameters
|* Parameter 1:     |sym1             ; sym1 corresponds to param1
|* parameter 2:     |sym2             ; sym2 corresponds to param2
|* parameter 3:     |                 ; Null string
```

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |THREE     |PARAM1,PARAM2,PARAM3
|_* PARAMETER 1:    |PARAM1           ; SYM1 corresponds to PARAM1
|_* PARAMETER 2:    |PARAM2           ; SYM2 corresponds to PARAM2
|_* PARAMETER 3:    |PARAM3           ; SYM3 corresponds to PARAM3
|THREE   |ENDM      |                 |
|        |LIST      |MAC              |
```

Call and expansion:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |THREE     |SYM1,SYM2,SYM3   |; Call matching prototype
|* PARAMETER 1:     |SYM1             ; SYM1 corresponds to PARAM1
|* PARAMETER 2:     |SYM2             ; SYM2 corresponds to PARAM2
|* PARAMETER 3:     |SYM3             ; SYM3 corresponds to PARAM3
```

3. The following examples demonstrate the use of the optional !.

   a. Macro BANG demonstrates the use of the embedded argument (1,2), syntactically valid expressions for positional-parameters ('abc,def') and keyword-parameters (PARAM3=1+2) and the null string.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |MACRO     |                 |
|        |BANG      |PARAM1,!PARAM2,!PARAM3=,PARAM4=
|_*PARAMETER 1:     |PARAM1           |; Embedded argument
|_*PARAMETER 2:     |PARAM2           |; Syntactically valid expression
|_*PARAMETER 3:     |PARAM3           |; Syntactically valid expression
|_*PARAMETER 4:     |PARAM4           |; Null string
|BANG    |ENDM      |                 |
|        |LIST      |MAC              |
```

Call and expansion:

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|        |BANG       |(1,2),'abc,def',PARAM3=1+2,, ; Macro call
|* PARAMETER 1:      1,2               ; Embedded argument
|* PARAMETER 2:      'abc,def'         ; Syntactically valid expression
|* PARAMETER 3:      1+2               ; Syntactically valid expression
|* PARAMETER 4:      |          Null string
```

- If the argument for PARAM1 had been (((1,2))), S1 would have gotten ((1,2)) at expansion.

- The ! specified on PARAM2 and PARAM3 permits commas and spaces to be embedded within strings 'abc,def' and allows expressions to be expanded without evaluation 1+2.

- PARAM4 passes a null string. A space or comma following the equal sign specifies a null or empty character string as the default argument.

b. The ! is remembered from the definition in macro remem.

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|        |macro      |                 |
|        |remem      |!param1=' 'r     |; Prototype statement includes !
|        |s1         |parma1           |
|remem   |endm       |                 |
|        |list       |mac              |
```

Call and expansion:

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|        |remem      |param1=' 'r      |; Macro call does not include !
|        |s1         |' 'r             |
```

4. The NULL and nullparm macros demonstrate the effect that null strings have when parameters are passed.

   a. NULL demonstrates the effect of a null string on macro expansions. P2 is passed a null string. When NULL is expanded, the resulting line is left-shifted two spaces; the difference between the length of the parameter (P2) and the null string.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |MACRO     |                 |
|        |NULL      |P1,P2,P3         |
|        |S1        |P1               |
|_*      |S2        |P2               |; Left-shifted two spaces
|        |S3        |P3               |
|NULL    |ENDM      |                 |
|        |LIST      |MAC              |
```

Call and expansion:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |NULL      |1,,3             |; Call
|        |S1        |1                |
|*       S2         |              ; Left-shifted three spaces
|        |S3        |3                |
```

b. Macro nullparm demonstrates how a macro is expanded when the macro call does not include the location field name specified on the macro definition.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |macro     |                 |
|        |nullparm  |longparm         |
|longparm|=         |1                |; Prototype statement
|nullparm|endm      |                 |
|        |list      |mac              |
```

Call and expansion:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |nullparm  |                 |
 1           1                ; prototype statement
```

NOTE

The location field parameter was omitted on the macro call in the previous example. The result and operand fields of the first line of the expansion were shifted left eight character positions because a null argument was substituted for the 8-character parameter, LONGPARM.

If the old format is used, only one space appears between the location field parameter and result field in the macro definition. If a null argument is substituted for the location parameter, the result field is shifted into the location field in column 2. Therefore, at least two spaces should always appear between a parameter in the location field and the first character in the result field in a definition.

If the new format is used, the result field is never shifted into the location field.

5. DEFAULT illustrates how defaults are assigned for keywords when the macro is expanded.

```
|Location|Result    |Operand              |Comment
|1       |10        |30                   |35
|        |          |                     |
|        |MACRO     |                     |
|        |DEFAULT   |PARAM1=(ABC DEF,GHI),PARAM2=ABC,PARAM3=
|_*   PARAM1        |                     |
|_*   PARAM2        |                     |
|_*   PARAM3        |                     |
|DEFAULT |ENDM      |                     |
|        |LIST      |MAC                  |
```

Calls and expansions:

```
|Location|Result     |Operand              |Comment
|1       |10         |30                   |35
|        |           |                     |
|        |DEFAULT    |PARAM1=ARG1,PARAM2=ARG2,PARAM3=ARG3 ; Macro call
|*   ARG1|           |                     |
|*   ARG2|           |                     |
|*   ARG3|           |                     |
|        |DEFAULT    |PARAM1=,PARAM2=(ARG2),PARAM3=ARG3 ; Macro call
|*       |           |                     |
|*   ARG2|           |                     |
|*   ARG3|           |                     |
|        |           |                     |
|        |DEFAULT    |PARAM1=((ARG1)),PARAM2=,PARAM3=ARG3 ; Macro call
|*   (ARG1)          |                     |
|*       |           |                     |
|*   ARG3|           |                     |
```

6. The following examples illustrate the correct and incorrect way to specify a literal string in a macro definition.

   a. WRONG illustrates the wrong way to specify a literal string in a macro definition. The comments in the expansion are writer comments and are not part of the expansion.

```
|Location|Result     |Operand              |Comment
|1       |10         |20                   |35
|        |           |                     |
|        |MACRO      |                     |
|        |WRONG      |PARAM1=' 'R          |; Prototype statement
|_*   PARAM1         |                     |
|WRONG   |ENDM       |                     |
|        |LIST       |MAC                  |; List expansion
```

Call and expansion (CAL erroneously expands WRONG; ' 'R was intended):

```
|Location|Result     |Operand              |Comment
|1       |10         |20                   |35
|        |           |                     |
|        |WRONG      |                     |; Call
|*   '   |           |                     |
```

   b. Macro right illustrates the right way to specify a literal string in a macro definition.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |macro     |                 |;
|        |right     |!param1=' 'r     |; Prototype statement
|_*   param1        |                 |
|right   |endm      |                 |
|        |list      |mac              |; List expansion
```

Expansion (CAL expands RIGHT as intended because of the !):

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |right     |                 |; Call
|*   ' 'r           |                 |
```

7. The following macros demonstrate the wrong and right methods for replacing parameters on the prototype statement with parameters on the macro call statement.

   a. BAD demonstrates the wrong method for replacing parameters.

```
|Location|Result    |Operand                  |Comment
|1       |10        |20                       |35
|        |          |                         |
|        |MACRO     |                         |
|        |BAD       |PARAM1,PARAM2,PARAM3=JJJ
|_* PARAMETER 1:    |PARAM1                   |
|_* PARAMETER 2:    |PARAM2                   |
|_* PARAMETER 3:    |PARAM3                   |
|BAD     |ENDM      |                         |
|LIST    |MAC       |                         |
```

Call and expansion:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |BAD       |PARAM3=KKK       |; Macro call
|* PARAMETER 1:     PARAM3=KKK        |
|* PARAMETER 2:     |                 |
|* PARMAETER 3:     JJJ               |
```

b. Macro good demonstrates the right method for replacing parameters.

```
|Location|Result    |Operand              |Comment
|1       |10        |20                   |35
|        |          |                     |
|        |macro     |                     |
|        |good      |param1,param2,param3=jjj
|_* parameter 1:    |param1               |; Null string
|_* parameter 2:    |param2               |; Null string
|_* parameter 3:    |param3               |
|good    |endm      |                     |
|        |list      |mac                  |
```

Call and expansion:

```
|Location|Result    |Operand              |Comment
|1       |10        |20                   |35
|        |          |                     |
|        |good      |,,param3=kkk         |; Macro call
|* parameter 1:     |          ; Null string
|* parameter 2:     |          ; Null string
|* parameter 3:     kkk                   |;
```

8. ALPHA demonstrates the specification of an embedded parameter.

```
|Location|Result    |Operand              |Comment
|1       |10        |20                   |35
|        |          |                     |
|        |MACRO     |                     |; EDIT=ON
|        |ALPHA     |!PARAM               |; Appending a string
|_* FORMAL PARM:    |PARAM                |
|_* EMBEDDED PARM:  |ABC_PARAM_DEFG       |; Concatenation off at call time
|ALPHA   |ENDM      |                     |
|        |LIST      |MAC                  |
```

Call and expansion:

```
|Location|Result    |Operand              |Comment
|1       |10        |20                   |35
|        |          |                     |
|        |ALPHA     |1                    |; Macro call
|* FORMAL PARM:     1                     |;
|* EMBEDDED PARM:   ABC1DEFG              |;
```

CAL processed the embedded parameter in macro ALPHA as follows:

1. CAL scans the string to identify the parameter. ABC_ cannot be a parameter, because the underscore character is not defined as an identifier character for a parameter.

2. CAL identifies PARAM as the parameter when the second underscore character is encountered.

3. 1 is substituted for PARAM producing the string ABC_1_DEFG.

4. If editing is enabled, the underscore characters are removed and the resulting string is ABC1DEFG.

   If editing is disabled, the string is ABC_1_DEFG.

5. CAL processes the statement.

5.12.3 OPDEF - OPERATION DEFINITION

An opdef (operation definition) identifies a sequence of statements that is called at a later point in the source program by a single instruction; the opdef call. Each time the opdef call occurs, the definition sequence is placed in the source program.

Opdefs resemble machine instructions and can be used to define new machine instructions or to redefine current machine instructions. Machine instructions map into opcodes that represent some hardware operation. When an operation is required that is not available through the hardware, an opdef can be written to perform that operation. When the opdef is called, the opdef maps into the opdef definition body and the operation is performed by the defined sequence specified in the definition body.

Any existing CAL machine instruction can be replaced with an opdef. Although opdef definitions should conform to meaningful operations that are supported by the hardware, they are not restricted to such operations.

The opdef definition sets up the parameters into which the arguments specified in the opdef call are substituted. Opdef parameters are always expressed in terms of registers or expressions. The opdef call passes arguments to the parameters in the opdef definition. Formal parameters can be specified in any form that is permitted by the BNF. The syntax for the opdef definition and the opdef call are identical with two exceptions:

- The complex register has been redefined for the opdef definition prototype statement as follows:

  complex-register ::=
    complex-register-mnemonic.*register-parameter*

- Expressions have been redefined for the opdef definition prototype statement as follows:

  expression ::= "@"[0 to 7 character expression-parameter]

These two exceptions allow parameters to be specified in the place of registers and expressions for an opdef definition.

The syntax defining a register-parameter and an expression-parameter is case sensitive. Every character identifying the parameter on the opdef prototype statement must identically match every character in the body of the opdef definition. This match includes the case (uppercase, lowercase, or mixed case) of each character.

Since the opdef can accept arguments in many forms, it can be more flexible than a macro. Opdefs place a greater responsibility for parsing arguments on the assembler. When a macro is specified, the responsibility for parsing arguments is placed on the user in many cases. Parsing a macro argument can involve numerous micro substitutions. These substitutions greatly increase the number of statements that are required to perform a similar operation with an opdef.

Defined sequences (macros, opdefs, dups, and echos) are costly in terms of assembler efficiency. As the number of statements in a defined sequence increases, the speed of the assembler decreases. This decrease in speed is directly related to the number of statements that are expanded and the number of times a defined sequence is called.

Limiting the number of statements in a defined sequence improves the performance of the assembler. In some cases, an opdef can perform the same operation that is performed by a macro and use fewer statements in the process.

The following example demonstrates that an opdef can accept many different kinds of arguments from the opdef call.

```
|Location|Result    |Operand         |Comment
|1       |10        |20              |35
|        |          |                |
|MANYCALL|OPDEF     |                |
|        |A.REG1    |A.REG2!A.REG3   |; Opdef prototype statement
|        |S1        |A.REG2          |
|        |S2        |A.REG3          |
|        |S3        |S1!S2           |
|        |A.REG1    |S3              |; Or of registers S1 and S2
|MANYCALL|ENDM      |                |
```

Calls and expansions:

```
|Location|Result      |Operand          |Comment
|1       |10          |20               |35
|        |            |                 |
|        |A1          |A2!A3            |; First call to opdef MANYCALL
|        |S1          |A.2              |
|        |S2          |A.3              |
|        |S3          |S1!S2            |
|        |A.1      S3                 ; Or of registers S1 and S2
|        |A.1         |A.2!A.3          |; Second call to Opdef MANYCALL
|        |S1          |A.2              |
|        |S2          |A.3              |
|        |S3          |S1!S2            |
|        |A.1      S3                 ; Or of registers S1 and S2
|        |A.ONE       |A.TWO!A.THREE    |; Third call to Opdef MANYCALL
|        |S1          |A.2              |
|        |S2          |A.3              |
|        |S3          |S1!S2            |
|        |A.ONE       S3               ; Or of registers S1 and S2
|        |A1          |A.2!A.THREE      |; Fourth call Opdef MANYCALL
|        |S1          |A.2              |
|        |S2          |A.3              |
|        |S3          |S1!S2            |
|        |A.1      S3                 ; Or of registers S1 and S2
```

In the first and second calls to opdef MANYCALL, the arguments passed to REG1, REG2, and REG3 are 1, 2, 3, respectively. In the third call to opdef MANYCALL, the arguments passed to REG1, REG2, and REG3 are ONE, TWO, and THREE, respectively. The fourth call to opdef MANYCALL demonstrates that the form of the arguments can vary within one call to an opdef as long as they take a form that is permitted by the BNF. The arguments passed REG1, REG2, and REG3 in the fourth call are 1, 2, and THREE, respectively.

The following example demonstrates how an opdef can be used to limit the number of statements required for a defined sequence.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |MACRO     |                 |
|        |$IF       |REG1,COND,REG2   |; Macro prototype statement
|        |.         |.                |
|        |.         |.                |
|        |.         |.                |
|$IF     |ENDM      |                 |
|        |.         |                 |
|        |.         |                 |
|        |.         |.                |
|        |$IF       |S6,EQ,S.3        |; Macro call
|        |.         |.                |
|        |.         |.                |
|        |.         |.                |
|        |$ELSE     |                 |
|        |.         |.                |
|        |.         |.                |
|        |.         |.                |
|        |$ENDIF    |                 |
```

Parsing the parameters (S6,EQ,S3) passed to the definition requires many micro substitutions within the definition body. These micros increase the number of statements within the definition body.

The same function is performed in the following example, but an opdef is specified instead of a macro. In this instance, specifying an opdef rather than a macro reduces the number of statements required for the function.

Since an opdef is called by its form, it is more flexible than a macro in accepting arguments. The opdef expects to be passed two S registers and the EQ mnemonic. The arguments for the registers can be specified in a number of ways and still be recognized as S register arguments by the opdef.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |opdef     |                 |
|example |$if       |s.reg1,eq,s.reg2 ; Opdef definition statement
|_* Register1:      |reg1             |
|_* Register2:      |reg2             |
|example |endm      |                 |
|        |list      |mac              |
```

Calls and expansions:

```
|Location|Result      |Operand          |Comment
|1       |10          |20               |35
|        |            |                 |
|        |$if         |s6,eq,s.3        |
|* Register1:       6                  |
|* Register2:       3                  |
```

An opdef is defined as global if it occurs within the global definitions part of a program segment. Opdef definitions are local if they occur within a program module (an IDENT, END sequence). A global definition can be redefined locally, but the global definition is reenabled and the local definition is discarded at the end of the program module. A global definition can be referenced anywhere within an assembler program after it has been defined.

The OPDEF pseudo instruction can be specified anywhere within a program segment. If the OPDEF pseudo instruction is found within a definition, it is defined. If the OPDEF pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

In the following example, the operand and comment fields of the expanded line are shifted two positions to the left (difference between reg and 1):

```
|Location|Result      |Operand          |Comment
|1       |10          |20               |35
|        |            |                 |
|example |opdef       |                 |
|        |s.reg       |@exp             |; Prototype statement
|        |a.reg       |@exp             |; New machine instruction
|example |endm        |                 |;
|        |list        |mac              |
```

Call and expansion:

```
|Location|Result      |Operand          |Comment
|1       |10          |20               |35
|        |            |                 |
|        |s1          |2                |; Opdef call
|        |a.1       2                  ; New machine instruction
```

5.12.3.1 Opdef definition

The OPDEF pseudo instruction is the first statement of an opdef definition. An opdef is constructed much like a macro. However, an opdef is defined not by a functional name like a macro but by the form of the opdef statement.

Opdef syntax is uniquely defined on the result field alone in which case the operand field is not specified or on the result and operand fields. The OPDEF prototype permits up to three subfields within the result and operand fields. At least 1 field must be present within the result field. No fields are required in the operand field.

The syntax for each of the subfields within the result and operand fields of the opdef prototype statement is identical. There are no special syntax forms for any of the subfields. The rules that apply for the first subfield in the result field apply to the remainder of the subfields within the result field and to all of the subfields within the operand field.

Format:

```
|Location|Result    |Operand          |Comment
|        |          |                 |
|name    |OPDEF     |                 |; OPDEF macro header
|[loc]   |defsynres |defsynop         |; Prototype statement
|        |LOCAL     |[name]{","[name]}| ; Optional LOCAL pseudos
|        |.         |.                |;
|        |.         |.                |; Definition body
|        |.         |.                |;
|name    |ENDM      |                 |; Definition end
```

```
|Location|Result    |Operand          |Comment
|        |          |                 |
|name    |opdef     |                 |; OPDEF macro header
|[loc]   |defsynres |defsynop         |; Prototype statement
|        |local     |[name]{","[name]}| ; Optional LOCAL pseudos
|        |.         |.                |;
|        |.         |.                |; Definition body
|        |.         |.                |;
|name    |endm      |                 |; Definition end
```

name OPDEF definition name. *name* identifies the definition and has no association with functionals appearing in the result field of instructions. *name* must match the name in the location field of the ENDM pseudo instruction, which ends the definition.

loc Optional location field parameter; *loc* must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

defsynres
Definition syntax for the result field; can be one, two, or three subfields specifying a valid result field syntax. The result field must be a symbolic.

Valid result subfields for opdefs can be one of the following:

- Initial-register
- Mnemonic
- Initial-expression

Initial-register; to specify an initial-register on the opdef prototype statement, use one of the four syntax forms for initial-registers described below:

initial-register ::=
[prefix] [register-prefix] register
[register-separator[register-ending]]

initial-register ::=
[prefix] [register-prefix] register
[register-expression-separator [register-ending]]

initial-register ::=
[prefix] [register-prefix] register
[register-expression-separator [expression-ending]]

initial-register ::=
[prefix] [register-prefix] register
[special-register-separator [register-ending]]

Optional prefix; can be "(" or "["

Optional register-prefix; case insensitive† and one of the following:

"<"	">"	"#<"	"#>"	"#"	"#F"	"#f"	"#H"
"#h"	"#I"	"#i"	"#P"	"#p"	"#Q"	"#q"	"#R"
"#r"	"#Z"	"#z"	"+"	"+F"	"+f"	"+H"	"+h"
"+I"	"+i"	"+P"	"+p"	"+Q"	"+q"	"+R"	"+r"
"+Z"	"+z"	"-"	"-F"	"-f"	"-H"	"-h"	"-I"
"-i"	"-P"	"-p"	"-Q"	"-q"	"-R"	"-r"	"-Z"
"-z"	"*"	"*F"	"*f"	"*H"	"*h"	"*I"	"*i"
"*P"	"*p"	"*Q"	"*q"	"*R"	"*r"	"*Z"	"*z"
"/"	"/F"	"/f"	"/H"	"/h"	"/I"	"/i"	"/P"
"/p"	"/Q"	"/q"	"/R"	"/r"	"/Z"	"/z"	"F"
"f"	"H"	"h"	"I"	"i"	"P"	"p"	"Q"
"q"	"R"	"r"	"Z"	"z"			

† When a register-prefix is specified on an opdef call, it is recognized by the opdef definition without regard to the case (uppercase or lowercase) in which it was entered.

Required register; a simple-register or a complex-register.

simple-register ::=
"CA" | "CE" | "CI" | "CL" | "MC" | "RT" | "SB" | "SM" | "VL" | "VM" | "XA"†

The complex-register has been redefined for the opdef prototype statement as follows:

register-designator ::=
complex-register-mnemonic.*register-parameter*

complex-register-mnemonic; case insensitive†† and one of the following:

"A" | "B" | "SB" | "SM" | "SR" | "ST" | "S" | "T" | "V"

register-parameter is a one to eight character identifier that is composed of identifier-characters.

optional register-separator; case insensitive†† and one of the following:

"+F" | "+f" | "+H" | "+h" | "+I" | "+i" | "+P" | "+p" | "+Q" | "+q" | "+R" | "+r" | "+Z" | "+z" | "-F" | "-f" | "-H" | "-h" | "-I" | "-i" | "-P" | "-p" | "-Q" | "-q" | "-R" | "-r" | "-Z" | "-z" | "*F" | "*f" | "*H" | "*h" | "*I" | "*i" | "*P" | "*p" | "*Q" | "*q" | "*R" | "*r" | "*Z" | "*z" | "/F" | "/f" | "/H" | "/h" | "/I" | "/i" | "/P" | "/p" | "/Q" | "/q" | "/R" | "/r" | "/Z" | "/z"

Optional register-expression-separator; can be one of the following:

")" | "]" | "&" | "!" | "\" | "#<" | "#>" | "<" | ">" | "+" | "-" | "*" | "/"

---

† When a simple-register or a complex-register-mnemonic is specified on an opdef call, it is recognized by the opdef definition without regard to the case (uppercase, lowercase, or mixed case) in which it was entered.

†† When a register-separator is specified on an opdef call, it is recognized by the opdef definition without regard to the case (uppercase or lowercase) in which it was entered.

Optional special-register-separator; specified as follows: "#"

Optional register-ending; to specify a register-ending, use one of the three syntax forms described below:

register-ending ::=
$register_1$ [register-separator [$register_2$ [suffix]]]

register-ending ::=
register [register-expression-separator [register-or-expression [suffix]]]

register-ending ::=
$register_1$ [special-register-separator [$register_2$ [suffix]]]

Required $register_1$; see register under initial-register.

Optional register-separator; see register-separator under initial-register.

Optional $register_2$; see register under initial-register.

Optional suffix; see suffix under initial-register.

Required register; see register under initial-register.

Optional register-expression-separator; see register-expression-separator under initial-register.

Optional register-or-expression; can be a register or an expression.

Required register if expression is not designated; see register under initial-register.

Required expression if register is not specified. Expression has been redefined for the opdef prototype statement as follows:

expression ::= *expression-parameter*

expression-parameter is an identifier that must begin with the @. The @ can be followed by from zero to seven identifier-characters.

special-register-separator; specified as follows: "#"

Optional expression-ending; specified as follows:

expression-ending ::=
expression [expression-separator [register-or-expression [suffix]]]

Required expression. Expression has been redefined for the opdef prototype statement as follows:

expression ::= *expression-parameter*

expression-parameter is an identifier that must begin with the @. The @ can be followed by from zero to seven identifier-characters.

Optional expression-separator; one of the following:

")" | "]" | "&" | "!" | "\" | "=" | "#<" | "#>"

Optional register-or-expression; can be a register or an expression.

Required register if expression is not designated; see register under initial-register.

Required expression if register is not specified; see expression.

Optional suffix; see suffix under initial-register.

Mnemonic; one to eight character identifier that must begin with a letter (A-Z or a-z), a decimal digit (0-9), or one of the following characters: $, %, &, ', *, +, -, ., /, :, =, ?, \, `, |, or ~. Optional characters 2 through 8 can be @ or any of the above-mentioned characters.

Initial-expression; to specify an initial-expression on the opdef prototype statement, use one of the following syntax forms for initial-expressions:

initial-expression ::=
[prefix] [expression-prefix] expression
[expression-separator [register-ending]]

initial-expression ::=
[prefix] [expression-prefix] expression
[expression-separator [expression-ending]]

initial-expression ::=
expression [expression-separator [register-ending]]

initial-expression ::=
expression [expression-separator [expression-ending]]

Optional prefix; one of the following: "(" or "["

Optional expression-prefix; one of the following:

"‹" | "›" | "#‹" | "#›"

Required expression; redefined for the opdef prototype statement as follows:

expression ::= **expression-parameter**

expression-parameter is an identifier that must begin with the @. The @ can be followed by from zero to seven identifier-characters.

Optional expression-separator; one of the following:

")" | "]" | "&" | "!" | "\" | "‹" | "›" | "#‹" | "#›"

Optional register-ending. See register-ending under initial-register.

Optional expression-ending. See expression-ending under initial-register.

defsynop Definition syntax for the operand field; can be zero, one, or two subfields specifying a valid operand field syntax. If a subfield exists in the result field, the first subfield in the operand field must be a symbolic.

The definition syntax for the operand field of an opdef is the same as the definition syntax for the result field of an opdef. See **defsynres** for a description of the definition syntax for subfields.

5.12.3.2 Opdef calls

An opdef definition is called by an instruction that matches the syntax of the result and operand fields as specified in the opdef prototype statement.

The arguments on the opdef call are passed to the parameters on the opdef prototype statement. The parameters on the opdef call can be entered in any form that is consistent with the BNF. The special syntax for registers and expressions that was required on the opdef definition does not extend to the opdef call. Anything that is permitted by the BNF is permitted on the opdef call.

Format:

Location	Result	Operand
loc-arg	*callsynres*	*callsynop*

locarg Optional location field argument. *locarg* must be terminated by a space. *locarg* can be any character up to but not including a space.

If a location field parameter is specified on the opdef definition. A matching location field parameter can be specified on the opdef call. *locarg* is substituted wherever the location field parameter occurs in the definition. If no location field parameter is specified in the definition, this field must be empty.

callsynres

Result field syntax for the opdef call. *callsynres* can consist of one, two, or three subfields and must have the same syntax as specified in the result field of the opdef definition prototype statement. The result field must be a symbolic as described in the BNF in appendix A, Instruction Syntax.

The syntax of the result field call is the same as the syntax of the result field definition with two exceptions. The special syntax rules that are in effect for registers and expressions on the opdef definition do not apply to the opdef call. The syntax for registers and expressions used on the opdef call is the same as the syntax for registers and expressions stated in the BNF.

The subfields in the result field on the opdef call can be specified with one of the following:

- Initial-register
- Mnemonic
- Initial-expression

For a description of the syntax for the result field of the opdef call, see the syntax for the result field of the opdef definition.

callsynop

Operand field syntax for the opdef call. *callsynop* can consist of zero, one, two, or three subfields and must have the same syntax as specified in the operand field of the opdef definition prototype statement. The operand field must be a symbolic as described in the BNF in appendix A, Instruction Syntax.

The syntax of the operand field call is the same as the syntax of the operand field definition with two exceptions. The special syntax rules that are in effect for registers and expressions on the opdef definition do not apply to the opdef call. The syntax for registers and expressions used on the opdef call is the same as the syntax for registers and expressions stated in the BNF.

The subfields in the operand field on the opdef call can be specified with one of the following:

- Initial-register
- Mnemonic
- Initial-expression

For a description of the syntax for the operand field of the opdef call, see the syntax for the result field of the opdef definition.

The following rules apply for opdef calls:

- The character strings *callsynres* and *callsynop* must be exactly as specified in the opdef definition.

- An expression must appear whenever an expression @exp is indicated in the prototype statement. The actual argument string is substituted in the definition sequence wherever the corresponding formal parameter @exp occurs.

- The actual argument string consisting of a complex-register-mnemonic followed by a "." followed by a register-parameter. A register-designator followed by a register-parameter must appear wherever the register-designator A.*register-parameter*, B.*register-parameter*, SB.*register-parameter*, S.*register-parameter*, T.*register-parameter*, ST.*register-parameter*, SM.*register-parameter*, or V.*register-parameter*, respectively, appeared in the prototype statement.

  - If the register-parameter is of the form octal-integer, the actual argument is the octal-integer part. The octal-integer is restricted to four octal digits.

  - If the register-parameter is of the form "." integer-constant or "." symbol, the actual argument is an integer-constant or a symbol.

Examples:

1. The following opdef definition illustrates a scalar floating-point divide sequence.

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|fdv     |opdef     |              |; Scalar floating-point
|L       |s.r1      |s.r2/fs.r3    |; divide prototype statement
|        |errif     |r1,eq,r2      |
|        |errif     |r1,eq,r3      |
|        |          |              |
|L       |s.r1      |/hs.r3        |
|        |s.r2      |s.r2*fs.r1    |
|        |s.r3      |s.r3*is.r1    |
|        |s.r1      |s.r2*fs.r3    |
|fdv     |endm      |              |
```

Opdef call and expansion:

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|a       |s4        |s3/fs2        |; Divide S3 by S2, result to S4
|        |errif     |4,eq,3        |
|        |errif     |4,eq,2        |
|a       |s.4       |/hs.2         |
|        |s.3       |s.3*fs.4      |
|        |s.2       |S.2*is.4      |
|        |s.4       |S.3*fs.2      |
```

2. The following opdef definition, call, and expansion define a conditional jump where a jump occurs if the A register values are equal.

Opdef definition:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|JEQ     |OPDEF     |                 |
|L       |JEQ       |A.A1,A.A2,@TAG   |; Opdef prototype statement
|L       |A0        |A_A1-A_A2        |
|_*      |JAZ       |@TAG             |; Expression is expected.
|JEQ     |ENDM      |                 |
|        |LIST      |MAC              |
```

Opdef call and expansion. The expansion starts on line 2.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |JEQ       |A3,A6,GO         |; Opdef call
|        A0         A3-A5             |
|*       JAZ        GO              ; Expression is expected.
```

3. The opdef in the following example demonstrates how an existing machine instruction can be redefined by an opdef.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|EXAMPLE |OPDEF     |                 |;
|        |S.REG     |@EXP             |; Opdef prototype instruction
|        |A.REG     |@EXP             |; New machine instruction
|EXAMPLE |ENDM      |                 |;
|        |LIST      |MAC              |;
```

Opdef call and expansion:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |S1        |2                |; Opdef call
|        A.1        2                 ; New machine instruction
```

4. The following example demonstrates how the expansion of an opdef is affected when the opdef call does not include a label that was specified in the opdef definition.

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|regchg  |opdef     |              |
|lbl     |s.reg1    |s.reg2        |; Opdef prototype statement
|lbl     |=         |*             |; Left-shifted if lbl is left off
|        |s.reg2    |s.reg1        |; Register S2 gets register S1
|regchg  |endm      |              |
|list    |mac       |              |
```

Opdef call and expansion:

```
|Location|Result    |Operand       |Comment
|1       |10        |20            |35
|        |          |              |
|        |s1        |s2            |; Opdef call
|     =            *              ; Left-shifted if lbl is left off
|      s.2          s.1           ; Register S2 gets register S1
```

NOTE

The location field parameter was omitted on the opdef call in the previous example. The result and operand fields of the first line of the expansion were shifted left three character positions because a null argument was substituted for the 3-character parameter, lbl.

If the old format is used, only one space appears between the location field parameter and result field in the macro definition. If a null argument is substituted for the location parameter, the result field is shifted into the location field in column 2. Therefore, at least two spaces should always appear between a parameter in the location field and the first character in the result field in a definition.

If the new format is used, the result field is never shifted into the location field.

5. The following example illustrates the case insensitivity of the register and register-prefix.

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|CASE    |OPDEF     |           |
|        |S1        |#Pa2       |; Prototype statement
|        |.         |.          |
|        |.         |.          |
|        |.         |.          |
|CASE    |ENDM      |           |
```

Opdef calls:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|        |S1        |#pa2       |; Recognized by CASE
|        |S1        |#Pa2       |; Recognized by CASE
|        |S1        |#pA2       |; Recognized by CASE
|        |S1        |#PA2       |; Recognized by CASE
|        |s1        |#pa2       |; Recognized by CASE
|        |s1        |#Pa2       |; Recognized by CASE
|        |s1        |#pA2       |; Recognized by CASE
|        |s1        |#PA2       |; Recognized by CASE
```

5.12.4 DUP - DUPLICATE CODE

The DUP pseudo instruction introduces the definition of a sequence of code that is assembled repetitively immediately following the definition. The dup sequence is assembled the number of times specified on the DUP pseudo instruction. The dup sequence to be repeated consists of statements following the DUP pseudo instruction and any optional LOCAL pseudo instructions. Comment statements are ignored. The dup sequence ends when the statement count is exhausted or when ENDDUP with a matching location field name is encountered.

Only one type of formal parameter is accepted within a DUP and must be specified with the LOCAL pseudo.

The DUP pseudo instruction can be specified anywhere within a program segment. If the DUP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the DUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*dupname*]	DUP	*expression*["," [*count*]]
[*dupname*]	dup	*expression*["," [*count*]]

dupname Optional name of the dup sequence; required if the count field is null or missing. *name* must match an ENDDUP name if no count field is present. The sequence field in the DUP pseudo itself represents the nested dup level and appears in columns 89 and 90 on the listing. For a description sequence field nest level numbering, see subsection 5.12.1, Similarities among defined sequences.

dupname must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

expression

An absolute expression with a positive value specifying the number of times to repeat the code sequence. All symbols, if any, must be previously defined. If the current base is mixed, octal is used for the expression. If the value is 0, the code is skipped. A STOPDUP can be used to override the given expression.

expression must meet the requirements for expressions as described in the BNF. For a description of expressions, see Expressions in subsection 4.7.

count Optional absolute expression with positive value specifying the number of statements to be duplicated. All symbols, if any, must be previously defined. If the current base is mixed, octal is used for the expression.

LOCAL pseudo instructions and comment statements (* in column 1) are ignored for the purpose of this count. Statements are counted before expansion of nested macro or opdef calls or dup or echo sequences.

count must meet the requirements for expressions as described in the BNF. For a description of expressions, see subsection 4.7.

Examples:

1. In the following example, the number of dups is 3 and the number of statements that are included in the dup definition is 5 :

```
|Location|Result      |Operand          |Comment
|1       |10          |20               |35
|        |            |                 |
|        |DUP         |3,5              |;
|        |LOCAL       |SYM1,SYM2        |; LOCAL pseudo; not counted
|* Asterisk comment; not counted        |
|        |S1          |1                |; First statement in definition
|* Asterisk comment; not counted        |
|        |INCLUDE     |ALPHA            |; INCLUDE pseudo; not counted
```

File ALPHA:

```
|Location|Result      |Operand          |Comment
|1       |10          |20               |35
|        |            |                 |
|        |S2          |3                |; Second statement in definition
|        |S4          |4                |; Third statement in definition
|* Asterisk comment; not included       |
|        |S5          |5                |; Fourth statement in definition
|        |S6          |6                |; Fifth statement in definition
```

2. The following two con pseudos are duplicated three times immediately following the definition.

```
|Location|Result      |Operand          |Comment
|1       |10          |20               |35
|        |            |                 |
|        |list        |dup              |
|example |dup         |3                |; Definition
|        |con         |1                |
|        |con         |2                |
|example |enddup      |                 |
```

Expansion:

```
|Location|Result      |Operand          |Comment
|1       |10          |20               |35
|        |            |                 |
|        |con         |1                |
|        |con         |2                |
|        |con         |1                |
|        |con         |2                |
|        |con         |1                |
|        |con         |2                |
```

5.12.5 ECHO - DUPLICATE CODE WITH VARYING ARGUMENTS

The ECHO pseudo instruction introduces the definition of a sequence of code that is assembled zero or more times immediately following the definition. On each repetition, the actual arguments are substituted for the formal parameters until the longest argument list is exhausted. Null strings are substituted for the formal parameters once shorter argument lists are exhausted. The echo sequence to be repeated consists of statements following the ECHO pseudo instruction and any optional LOCAL pseudo instructions. Comment statements are ignored. The echo sequence ends with an ENDDUP that has a matching location field name.

The STOPDUP pseudo instruction can be used to override the repetition count determined by the number of arguments in the longest argument list.

The ECHO pseudo instruction can be specified anywhere within a program segment. If the ECHO pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ECHO pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
dupname	ECHO	[*name*"="*argument*]{","[*name*"="*argument*]}
dupname	echo	[*name*"="*argument*]{","[*name*"="*argument*]}

dupname Required name of the echo sequence. *dupname* must match the location field name in the ENDDUP instruction that terminates the echo sequence.

dupname must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

name Formal parameter name; must be unique. There can be none, one, or more formal parameters.

name must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

argument List of actual arguments. The list can be a single argument or a parenthesized list of arguments.

A single argument is any ASCII character up to but not including the element separator, a space, or a semicolon (new format only). The first character cannot be a left parenthesis.

A parenthesized list can be a list of one or more actual arguments. Each actual argument can be one of the following:

- An ASCII character string can contain embedded arguments. If, however, an ASCII string is intended the first character in the string cannot be a left parenthesis. The following is a legal ASCII string: 4(5)

  The following is an illegal ASCII string: (5)4(5)

- A null argument; an empty ASCII character string.

- An embedded-argument that contains a list of arguments enclosed in matching parentheses. An embedded argument can contain blanks or commas and matched pairs of parentheses. The outermost parentheses are always stripped from an embedded argument when an echo definition is expanded.

  An embedded-argument must meet the requirements for embedded arguments as described in the BNF. For a description of embedded-arguments, see subsection 4.7, Expressions.

Examples:

1. In the following example, the ECHO pseudo is expanded twice immediately followng the definition:

```
|Location|Result    |Operand              |Comment
|1       |10        |20                   |35
|        |          |                     |
|        |LIST      |DUP                  |
|EXAMPLE |ECHO      |PARAM=(1,3),PARAM=(2,4) ; Definition
|        |CON       |PARAM1               |; Gets 1 and 3
|        |CON       |PARAM2               |; Gets 2 and 4
|EXAMPLE |ENDDUP    |                     |
```

Expansion:

```
|Location|Result    |Operand              |Comment
|1       |10        |20                   |35
|        |          |                     |
|        |CON       |1                    |; Gets 1 and 3
|        |CON       |2                    |; Gets 2 and 4
|        |CON       |3                    |; Gets 1 and 3
|        |CON       |4                    |; Gets 2 and 4
```

2. In the following example, the echo pseudo is expanded once immediately following the definition with 2 null arguments.

```
|Location|Result     |Operand              |Comment
|1       |10         |20                   |35
|        |           |                     |
|        |list       |dup                  |
|example |echo       |param1=,param2=() ; ECHO with two null
|        |           |                     |; parameters
|_* Parameter 1 is:  'param1'              |
|_* Parameter 2 is:  'param2'              |
|example |enddup     |                     |
```

Expansion:

```
|Location|Result     |Operand              |Comment
|1       |10         |20                   |35
|        |           |                     |
|* Parameter 1 is:  ''                     |
|* Parameter 2 is:  ''                     |
```

5.12.6 ENDM - END MACRO OR OPDEF DEFINITION

The body of a macro or opdef definition is terminated by an ENDM pseudo instruction. ENDM has no effect if used within a MACRO or OPDEF definition with a different name.

The ENDM pseudo instruction can only be specified within a macro or opdef definition. If the ENDM pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result     |Operand
|        |           |
|func    |ENDM       |ignored
|func    |endm       |ignored
```

func Name of the macro or opdef definition sequence, must be a valid identifier or the equal sign. *func* must match the functional appearing in the result field of the macro prototype or the location field name in an OPDEF instruction.

If the ENDM pseudo instruction is encountered within a definition but *func* does not match the name of an opdef or the functional of a macro, the ENDM instruction is defined and does not terminate the opdef or macro definition in which it is found.

func must meet the requirements for functionals as described in the BNF. For a description of functionals, see appendix A, Instruction Syntax.

5.12.7 EXITM - PREMATURE EXIT OF A MACRO EXPANSION

The EXITM pseudo immediately terminates the innermost nested macro or opdef expansion, if any, caused by either a macro or an opdef call. If files were included within this expansion and/or one or more dup or echo expansions are in progress within the innermost macro or opdef expansion, they will also be terminated immediately. If such an expansion does not exist, the EXITM pseudo issues a caution level listing message and does nothing.

The EXITM pseudo instruction can be specified anywhere within a program segment. If the EXITM pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the EXITM pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
ignored	EXITM	ignored
ignored	exitm	ignored

Example:

In the following macro call, the macro expansion is terminated immediately as a result of the EXITM pseudo. Therefore, the second comment is not included as part of the expansion.

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|        |macro      |                 |
|        |alpha      |                 |
|_*  First comment   |                 |
|        |exitm      |                 |
|_*  Second comment  |                 |
|alpha   |endm       |                 |
|        |list       |mac              |
```

Call and expansion:

```
|Location|Result     |Operand          |Comment
|1       |10         |20               |35
|        |           |                 |
|        |alpha      |                 |; Macro call
|*  First comment    |                 |
|        |exitm      |                 |
```

5.12.8 ENDDUP - END DUPLICATED CODE

The ENDDUP pseudo instruction ends the definition of the code sequence to be repeated. An ENDDUP pseudo instruction terminates a dup or echo definition with the same name. ENDDUP has no effect if used within a DUP or ECHO definition with a different location field name. ENDDUP has no effect on a dup definition that is terminated by a statement count.

The ENDDUP pseudo instruction is restricted to definitions (DUP or ECHO). If the ENDDUP pseudo instruction is found on a MACRO or OPDEF definition, it is defined and is not recognized as a pseudo instruction. If the ENDDUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result     |Operand
|        |           |
|dupname |ENDDUP     |ignored
|dupname |enddup     |ignored
```

dupname Required name of a dup sequence. *name* must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

5.12.9 NEXTDUP - PREMATURE EXIT OF THE CURRENT ITERATION OF A DUPLICATION EXPANSION

The NEXTDUP pseudo stops the current iteration of a duplication sequence indicated by a DUP or an ECHO pseudo instruction. Assembly of the current repitition of the dup sequence is terminated immediately and the next repitition, if any, is begun.

Assembly of the current iteration of the innermost duplication expansion with a matching location field name is terminated immediately; however, if the location field name is not present, assembly of the current iteration of the innermost duplication expansion is terminated immediately.

If other dup, echo, macro, or opdef expansions were included within the duplication expansion to be terminated, these expansions are also terminated immediately. In addition, if a file is being included at expansion time within the duplication expansion to be terminated, the inclusion of that file is terminated immediately.

The NEXTDUP pseudo instruction can be specified anywhere within a program segment. If the NEXTDUP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the NEXTDUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo.

Format:

Location	Result	Operand
[*dupname*]	NEXTDUP	ignored
[*dupname*]	nextdup	ignored

dupname Name of a dup sequence. If the name is present but does not match any existing duplication expansion, a caution level listing messages is issued and the pseudo does nothing. If the name is not present and a duplication expansion does not currently exist, a caution level listing message is issued and the pseudo does nothing.

5.12.10 STOPDUP - STOP DUPLICATION

The STOPDUP pseudo instruction stops duplication of a code sequence indicated by a DUP or ECHO pseudo instruction. STOPDUP overrides the repetition count. Assembly of the current dup sequence is terminated immediately. STOPDUP terminates the innermost dup or echo sequence with the same name as found in the location field. If there is no location

field name, STOPDUP will terminate the innermost dup or echo sequence. STOPDUP does not affect the definition of the code sequence to be duplicated.

Assembly of the innermost duplication expansion with a matching location field name is terminated immediately; however, if the location field name is not present, assembly of the innermost duplication expansion is terminated immediately. If other dup, echo, macro, or opdef expansions were included within the duplication expansion to be terminated, these expansions are also terminated immediately. In addition, if a file is being included at expansion time within the duplication expansion to be terminated, the inclusion of that file is terminated immediately.

The STOPDUP pseudo instruction can be specified anywhere within a program segment. If the STOPDUP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the STOPDUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

Location	Result	Operand
[*dupname*]	STOPDUP	ignored
[*dupname*]	stopdup	ignored

dupname Name of a dup sequence. If the name is present but does not match any existing duplication expansion, a caution level listing message is issued and the pseudo does nothing. If the name is not present and a duplication expansion does not currently exist, a caution level listing message is issued and the pseudo does nothing.

name must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

Examples:

1. The following example uses a DUP pseudo instruction to define an array with values 0, 1, and 2.

Location	Result	Operand	Comment
1	10	20	35
S	=	W.*	
	DUP	3,1	
	CON	W.*-S	

DUP expansion:

```
|Location|Result    |Operand      |Comment
|1       |10        |20           |35
|        |          |             |
|        |CON       |W.*-S        |
|        |CON       |W.*-S        |
|        |CON       |W.*-S        |
```

2. The ECHO and DUP pseudo instructions define a nested duplication in the following example.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|ECHO    |ECHO      |RI=(A,S),RJK=(B,T)
|I       |SET       |0                |
|DUPI    |DUP       |8                |
|JK      |SET       |0                |
|DUPJK   |DUP       |64               |
|        |RI.I      |RJK.JK           |
|        |          |                 |
|JK      |SET       |JK+1             |
|DUPJK   |ENDDUP    |                 |
|I       |SET       |I+1              |
|DUPI    |ENDDUP    |                 |
|ECHO    |ENDDUP    |                 |
```

ECHO and DUP expansion. (The following expansion is not generated by CAL, but is included to illustrate the expansion of the previously nested duplication expansion.)

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|        |          |          |; In the first call of the echo,
|        |          |          |; the A and B parameters are
|        |          |          |; used.
|        |A.0       |B.0       |; DUPJK generates the register
|        |.         |.         |; A.0 gets register B.0 through
|        |.         |.         |; register A.0 gets register
|        |.         |.         |; B.64 instructions.
|        |A.0       |B.64      |;
|        |.         |.         |; DUPI increments the
|        |.         |.         |; A register from A.1 to A.7
|        |.         |.         |; for succeeding passes through
|        |A.1       |B.0       |; DUPJK.  DUPJK generates
|        |.         |.         |; register A.i gets register
|        |.         |.         |; B.0 through register A.i
|        |.         |.         |; gets register B.64
|        |A.7       |B.64      |; instructions; where i is 1 to 7
|        |S.0       |T.0       |; In the second expansion of the
|        |.         |.         |; echo pseudo , the S and T
|        |.         |.         |; parameters are used.
|        |.         |.         |;
|        |S.0       |T.64      |; DUPJK and DUPI generate the
|        |.         |.         |; same series of register
|        |.         |.         |; instructions for the S and T
|        |.         |.         |; registers that were generated
|        |          |          |; for the A and B registers.
|        |S.8       |T.64      |
```

3. The STOPDUP pseudo instruction terminates duplication.

```
|Location|Result    |Operand   |Comment
|1       |10        |20        |35
|        |          |          |
|        |LIST      |DUP       |
|T       |SET       |0         |
|A       |DUP       |1000      |
|T       |SET       |T+1       |
|        |IFE       |T,EQ,3,1  |; Terminate duplication when T=3
|A       |STOPDUP   |          |
|        |CON       |T         |
|A       |ENDDUP    |          |
```

Expansion:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|T       |SET       |T+1        |
|        |CON       |T          |
|T       |SET       |T+1        |
|        |CON       |T          |
|T       |SET       |T+1        |
|A       |STOPDUP   |           |
```

4. The following example uses a STOPDUP pseudo instruction to immediately terminate a DUP.

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|DNAME   |DUP       |3          |
|_*  First comment  |           |
|        |STOPDUP   |           |
|_*  Second comment |           |
|DNAME   |ENDUP     |           |
```

Expansion:

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|*  First comment   |           |
|        |STOPDUP   |           |
```

5. The following example is similar to example four except that in this example, NEXTDUP replaces STOPDUP. The current iteration is terminated immediately when the NEXTDUP pseudo is encountered.

```
|Location|Result    |Operand    |Comment
|1       |10        |20         |35
|        |          |           |
|DNAME   |DUP       |3          |
|_*  First comment  |           |
|        |NEXTDUP   |           |
|_*  Second comment |           |
|DNAME   |ENDUP     |           |
```

Expansion:

```
|Location|Result      |Operand          |Comment
|1       |10          |20               |35
|        |            |                 |
|*  First comment     |                 |
|        |NEXTDUP     |                 |
|*  First comment     |                 |
|        |NEXTDUP     |                 |
|*  First comment     |                 |
|        |NEXTDUP     |                 |
```

5.12.11 LOCAL - SPECIFY LOCAL UNIQUE CHARACTER STRING

The LOCAL pseudo instruction specifies unique character string replacements within a program segment that are defined only within the macro, opdef, dup, or echo definition. These character string replacements are only known in the macro, opdef, dup, or echo at expansion time. The most common usage of the LOCAL pseudo is for defining symbols, but the LOCAL pseudo is not restricted to the definition of symbols. Local pseudos within a macro, opdef, dup, or echo header are not part of the macro definition.

On each macro/opdef call and each repetition of a dup or echo definition sequence, the assembler creates a unique 8-character string (commonly used for the definition of symbols by the user) for each local parameter and substitutes the created string for the local parameter on each occurrence within the definition. The unique character string created for local parameters has the form *%%nnnnnn*, where *n* is a decimal digit.

Zero or more LOCAL pseudo instructions can appear in the header of a macro, opdef, dup, or echo definition. The LOCAL pseudo instructions must immediately follow the macro or opdef prototype statement or DUP or ECHO pseudo instructions, except for intervening comment statements.

The LOCAL pseudo instruction can only be specified within a definition. If the LOCAL pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result      |Operand
|        |            |
|ignored |LOCAL       |[name]{","[name]}
|ignored |local       |[name]{","[name]}
```

name Formal parameters that must be unique and are to be rendered local to the definition. *name* must meet the requirements for names as described in the BNF. For a description of names, see subsection 4.2.

Examples:

1. The following example demonstrates that all formal parameters must be unique.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |MACRO     |                 |
|        |UNIQUE    |PARM2            |; PARM2 is defined within UNIQUE
|        |LOCAL     |PARM1,PARM2      |; Error; PARM2 previously
|        |.         |.                |; defined as a parameter in the
|        |.         |.                |; macro prototype statement.
|        |.         |.                |
|UNIQUE  |ENDM      |                 |
```

2. The following example demonstrates how a unique character string is generated for each parameter defined by the local pseudo.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |macro     |                 |
|        |string    |                 |;
|        |local     |param1,param2    |; Not part of the definition
|        |          |                 |; body
|param1  |=         |1                |;
|        |s1        |param1           |; Register s1 gets the value
|        |          |                 |; defined by param1.
|        |          |                 |
|param2  |=         |2                |;
|        |s2        |param2           |; Register s2 gets the value
|        |          |                 |; defined by param2.
|string  |endm      |                 |;
|        |list      |mac              |;
```

Call and expansion:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |string    |                 |;
|%%262144|  =       |  1              |;
|        |s1        |%%262144         |  ; Register s1 gets the value
|        |          |                 |; defined by param1.
|%%131072|  =       |  2              |;
|        |s2        |%%131072         |  ; Register s2 gets the value
|        |          |                 |; defined by param2.
```

The call to the macro string generates unique strings for param1 (%%262144) and for param2 (%%131072).

5.12.12 OPSYN - SYNONYMOUS OPERATION

The OPSYN pseudo instruction defines an operation that is synonomous with another macro or pseudo operation. The functional in the location field is defined as being the same as the functional in the operand field. Any pseudo instruction or macro can be redefined in this manner.

The functional in the location field can be a currently defined macro or pseudo in which case the current definition is replaced and a message is issued informing you that a redefinition has occurred.

An operation defined by OPSYN is global if the OPSYN pseudo occurs within the global part of an assembler segment, and it is local if the OPSYN pseudo appears within an assembler module of a segment. Global operations can be referenced in any program segment following the definition. Every local operation is removed at the end of a program module, making any previous global definition with the same name available again.

If the OPSYN pseudo instruction occurs within a definition, it is defined and is not recognized as a pseudo instruction. IF the OPSYN pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

Format:

```
|Location|Result    |Operand
|        |          |
|func1   |OPSYN     |[func2]
|func1   |opsyn     |[func2]
```

func1 Required functional name; a valid functional. The name of a defined operation such as a pseudo instruction or macro, or the equal sign. *func1* must not be blank.

func1 must meet the requirements for functionals as described in the BNF. For a description of functionals, see appendix A, Instruction Syntax.

func2 Optional functional name The name of a defined operation or the equal sign. If *func2* is blank, *func1* becomes a do-nothing pseudo instruction.

Examples:

1. The following macro definition includes the OPSYN pseudo instruction that redefines the IDENT pseudo instruction.

   OPSYN definition:

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|IDENTT  |OPSYN     |IDENT            |
|        |MLEVEL    |ERROR            |; Eliminates the warning error
|        |          |                 |; that is issued because the
|        |          |                 |; IDENT pseudo is redefined
|        |MACRO     |                 |
|        |IDENT     |NAME             |
|        |LIST      |LIS,OFF,NXRF     |
|NAME    |LIST      |LIS,ON,XRF       |; Processed if LIST=NAME on CAL
|        |          |                 |; control statement
|        |IDENTT    |NAME             |
|IDENT   |ENDM      |                 |
```

   OPSYN call and expansion. The expansion starts on line 2.

```
|Location|Result    |Operand          |Comment
|1       |10        |20               |35
|        |          |                 |
|        |IDENT     |A                |
|        |LIST      |LIS,OFF,NXRF     |
|A     LIST        LIS,ON,XRF        |; Process if LIST=NAME on CAL
|        |          |                 |; control statement
|        |IDENTT    |A                |
```

2. Macro first illustrates that a functional can be redefined a number of times.

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |macro     |            |
|        |first     |            |
|        |s1        |1           |
|        |s2        |2           |
|        |s3        |s1+s2       |
|first   |endm      |            |
|second  |opsyn     |first       |; second is the same as first
|third   |opsyn     |second      |; third is the same as second
```

Opdef calls and expansions:

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|        |first     |            |; Macro call
|        |s1        |1           |
|        |s2        |2           |
|        |s3        |s1+s2       |
|        |second    |            |
|        |s1        |1           |
|        |s2        |2           |
|        |s3        |s1+s2       |
|        |third     |            |
|        |s1        |1           |
|        |s2        |2           |
|        |s3        |s1+s2       |
```

3. In the following example the functional EQU is defined to perform the same operation as =.

```
|Location|Result    |Operand     |Comment
|1       |10        |20          |35
|        |          |            |
|EQU     |OPSYN     |=           |; EQU is defined to perform the
|        |          |            |; operation that the = pseudo
|        |          |            |; instruction performs.
```

APPENDIX SECTION

A. INSTRUCTION SYNTAX

This appendix lists the CAL instruction syntax that describes machine instructions and opdefs. The syntax for pseudos and macros are described in section 5, Pseudo Instructions. The instructions themselves are described in one of the following appropriate symbolic machine instruction manuals:

- Symbolic Machine Instructions Reference Manual, CRI publication SR-0085
- CRAY-2 Computer System Functional Description, publication HR-2000.

Opdefs and the OPDEF pseudo instruction are described in subsection 5.12.3, OPDEF - Operation Definition.

A.1 INSTRUCTION SYNTAX CONVENTIONS

Each machine instruction and opdef can be defined syntactically using the Backus-Naur Form (BNF). BNF is a hierarchy of definitions. The BNF conventions used in this manual are described under Conventions in section 1, Introduction.

A.2 CAL INSTRUCTION SYNTAX

This section contains the rules for the syntax of a symbolic machine instruction or opdef, and a description of how the syntax is used. This section also contains an alphabetical listing of all the symbols used in the Backus-Naur Form for a CAL instruction.

A.2.1 SYNTAX DESCRIPTION

BNF requires a starting point. For the CAL syntax, the result field is the starting point. The statement

```
result field ::= functional | symbolic .
```

identifies the result field as a nonterminal symbol that can be expanded to include either functional or symbolic. Both functional and symbolic are nonterminal symbols.

The second statement

```
functional ::= identifier | "=" .
```

identifies functional as a nonterminal symbol that can be expanded to include either the nonterminal symbol identifier or the terminal symbol "=".

The translation continues in this manner until every nonterminal symbol is expanded to include only terminal symbols.

Generally, CAL evaluates the operand field of an instruction in the same way that it evaluates the result field. If the result field of an instruction is a functional (pseudos and macros), the operand field syntax is dependent on the function. If the result field of an instruction is a symbolic (machine instruction or opdef), the operand field of the instruction must also be a symbolic.

A.2.2 INSTRUCTION SYNTAX (HIERARCHICAL VERSION)

The CAL instruction syntax begins with the result field and works through all of the possible instructions.

NOTE

CAL uses the BNF rules while parsing statements. Parsing is done statement by statement. Ambiguities are resolved by scanning BNF productions in a left to right order and accepting the first valid production that matches currently defined pseudo, macro, machine, or opdef instructions.

The following symbolic instruction syntax is presented in hierarchical order.

```
result-field ::= functional | symbolic .

functional ::= identifier | "=" .
```

```
symbolic ::=
    [ subfield ] [ subfield-separator [ subfield ]
    [ subfield-separator [ subfield ] ] ] .

subfield-separator ::= "," .

subfield ::=
    initial-register | mnemonic | initial-expression .

initial-register ::=
    [ prefix ] [ register-prefix ] register
    [ register-separator [ register-ending ] ] |
    [ prefix ] [ register-prefix ] register
    [ register-expression-separator
    [ register-ending ] ] |
    [ prefix ] [ register-prefix ] register
    [ register-expression-separator
    [ expression-ending ] ] |
    [ prefix ] [ register-prefix ] register
    [ special-register-separator [ register-ending ] ] .

initial-expression ::=
    [ prefix ] [ expression-prefix ] expression
    [ expression-separator [ register-ending ] ] |
    [ prefix ] [ expression-prefix ] expression
    [ expression-separator [ expression-ending ] ] |
    expression [ expression-separator
    [ register-ending ] ] |
    expression [ expression-separator
    [ expression-ending ] ] .

register-ending ::=
    register [ register-separator [ register [ suffix ] ] ] |
    register [ register-expression-separator
    [ register-or-expression [ suffix ] ] ] |
    register [ special-register-separator
    [ register [ suffix ] ] ] .

expression-ending ::=
    expression [ expression-separator
    [ register-or-expression [ suffix ] ] ] .

prefix ::= "(" | "[" .

suffix ::= ")" | "]" .

register-prefix ::=
    expression-prefix |
    complement-character [ prefixed-register-character ] |
    arithmetic-character [ prefixed-register-character ] |
    prefixed-register-character .
```

```
expression-prefix ::=
    prefixed-expression-character |
    complement-character prefixed-expression-character .

register-separator ::=
    arithmetic-character prefixed-register-character .

special-register-separator ::= complement-character .

register-expression-separator ::=
    expression-separator | arithmetic-character .

expression-separator ::=
    suffix | logical-character | "=" |
    [ complement-character ] prefixed-expression-character .

prefixed-register-character ::=
    "F" | "f" | "H" | "h" | "I" | "i" | "P" | "p" |
    "Q" | "q" | "R" | "r" | "Z" | "z" .

prefixed-expression-character ::= "<" | ">" .

complement-character ::= "#" .

arithmetic-character ::= "+" | "-" | "*" | "/" .

logical-character ::= "&" | "!" | "\" .

register-or-expression ::= register | expression .

register ::= complex-register | simple-register .

complex register ::=
    complex-register-mnemonic register-designator .

simple-register ::= simple-register-mnemonic .

complex-register-mnemonic ::=†
    "A" | "B" | "SB" | "SM" | "SR" | "S" | "T" | "ST" | "V" .

simple-register-mnemonic ::=†
    "CA" | "CE" | "CI" | "CL" | "MC" | "RT" | "SB" | "SM" | "VL" | "VM" |
    "XA" .

register-designator ::=
    octal-digit [ octal-digit [ octal-digit [ octal-digit ] ] ] |
    "." integer-constant |
    "." symbol .
```

---

† All characters within these terminal symbols can be either uppercase or lowercase.

```
symbol ::= unqualified-symbol | qualified-symbol .

unqualified-symbol ::= identifier .

qualified-symbol ::= "/" [ identifier ] "/" identifier .

identifier ::=
    initial-identifier-character
    [ identifier-character [ identifier-character
    [ identifier-character [ identifier-character
    [ identifier-character [ identifier-character
    [ identifier-character ] ] ] ] ] ] ] .

initial-identifier-character ::= letter | "$" | "%" | "@" .

identifier-character ::=
    initial-identifier-character | decimal-digit .

letter ::=
    "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
    "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
    "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
    "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
    "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
    "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" .

octal-digit ::=
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" .

decimal-digit ::= octal-digit | "8" | "9" .

hex-digit ::=
    decimal-digit |
    "A" | "a" | "B" | "b" | "C" | "c" | "D" | "d" |
    "E" | "e" | "F" | "f" .

expression ::=
    embedded-argument |
    [ add-operator ] term { add-operator term } .

embedded-argument ::= "(" { argument } ")" .

argument ::=
    argument-character | embedded-argument .
```

```
argument-character ::=
    " " | "!" | """ | "#" | "$" | "%" | "&" | "'" | "*" |
    "+" | "," | "-" | "." | "/" | "0" | "1" | "2" | "3" |
    "4" | "5" | "6" | "7" | "8" | "9" | ":" | ";" | "<" |
    "=" | ">" | "?" | "@" | "A" | "B" | "C" | "D" | "E" |
    "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" |
    "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" |
    "X" | "Y" | "Z" | "[" | " " | "]" | "^" | "_" | "`" |
    "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
    "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
    "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "{" |
    "|" | "}" | "~" .

term ::=
    prefixed-element { multiply-operator prefixed-element } .

add-operator ::= "+" | "-" .

multiply-operator ::= "*" | "/" .

prefixed-element ::= [ "#" ] [ element-prefix ] element .

element-prefix ::= "P." | "p." | "W." | "w." .

element ::= special-element | constant | symbol | literal .

special-element ::=
    "*A" | "*a" | "*B" | "*b" | "*O" | "*o" | "*P" | "*p" | "*W" | "*w" |
    "*" .

constant ::=
    floating-constant | integer-constant |
    character-constant .

floating-constant ::=
    [ decimal-prefix ] floating-decimal
    [ binary-scale decimal-integer ] ] .

floating-decimal ::=
    decimal-integer decimal-fraction
    [ decimal-exponent decimal-integer ] |
    decimal-integer "."
    [ decimal-exponent decimal-integer ] |
    decimal-integer decimal-exponent decimal-integer |
    decimal-fraction [ decimal-exponent decimal-integer ] .

decimal-exponent ::=
    "E" [ add-operator ] | "e" [ add-operator ] |
    "D" [ add-operator ] | "d" [ add-operator ] .

decimal-fraction ::= "." decimal-integer .
```

```
integer-constant ::=
    base-integer [ binary-scale base-integer ] |
    octal-prefix octal-integer
    [ binary-scale octal-integer ] |
    decimal-prefix decimal-integer
    [ binary-scale decimal-integer ] |
    hex-prefix hex-integer [ binary-scale hex-integer ] .

base-integer ::= decimal-integer .

octal-prefix ::= "O'" | "o'" .

decimal-prefix ::= "D'" | "d'" .

hex-prefix ::= "X'" | "x'" .

octal-integer ::= octal-digit { octal-digit } .

decimal-integer ::= decimal-digit { decimal-digit } .

hex-integer ::= hex-digit { hex-digit } .

binary-scale ::=
    "S" [ add-operator ] | "s" [ add-operator ] .

character-constant ::=
    [ character-prefix ] character-string
    [ character-suffix ] .

character-prefix ::= "A" | "a" | "C" | "c" | "E" | "e" .

character-string ::= "'" { string-character } "'" .

string-character ::=
    "'" | " " | "!" | """ | "#" | "$" | "%" | "&" | "(" |
    ")" | "*" | "+" | "," | "-" | "." | "/" | "0" | "1" |
    "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | ":" |
    ";" | "<" | "=" | ">" | "?" | "@" | "A" | "B" | "C" |
    "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" |
    "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
    "V" | "W" | "X" | "Y" | "Z" | "[" | "\" | "]" | "^" |
    "_" | "`" | "a" | "b" | "c" | "d" | "e" | "f" | "g" |
    "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
    "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" |
    "z" | "{" | ">" | "}" | "~" .

character-suffix ::=
    "H" | "h" | "L" | "l" | "R" | "r" | "Z" | "z" .
```

```
literal ::= "=" data-item .

data-item ::=
    floating-data | integer-data | character-data .

floating-data ::= [ sign ] floating-constant .

integer-data ::= [ sign ] integer-constant .

sign ::= "+" | "-" | "#" .

character-data ::=
    [ character-prefix ] character-string
    [ character-count ] [ character-suffix ] .

character-count ::= base-integer | prefixed-integer | "*" .

prefixed-integer ::=
    octal-prefix octal-integer |
    decimal-prefix decimal-integer |
    hex-prefix hex-integer .

mnemonic ::=
    initial-mnemonic-character
    [ mnemonic-character [ mnemonic-character
    [ mnemonic-character [ mnemonic-character
    [ mnemonic-character [ mnemonic-character
    [ mnemonic-character ] ] ] ] ] ] ] .

initial-mnemonic-character ::=
    letter | decimal-digit | "$" | "%" | "&" | "'" | "*" | "+" | "-" |
                "." | "/" | ":" | "=" | "?" | "\" | "`" | "|" | """ .

mnemonic-character ::= initial-mnemonic-character | "@" .
```

A.2.3 INSTRUCTION SYNTAX (SORTED VERSION)

The CAL instruction syntax begins with the result field and works through all the possible instructions. The following syntax is presented in alphabetical order.

```
add-operator ::= "+" | "-" .

argument ::= argument-character | embedded-argument .
```

```
argument-character ::=
    " " | "!" | """ | "#" | "$" | "%" | "&" | "'" | "*" |
    "+" | "," | "-" | "." | "/" | "0" | "1" | "2" | "3" |
    "4" | "5" | "6" | "7" | "8" | "9" | ":" | ";" | "<" |
    "=" | ">" | "?" | "@" | "A" | "B" | "C" | "D" | "E" |
    "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" |
    "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" |
    "X" | "Y" | "Z" | "[" | " " | "]" | "^" | "_" | "`" |
    "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
    "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
    "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "{" |
    "|" | "}" | "~" .

arithmetic-character ::= "+" | "-" | "*" | "/" .

base-integer ::= decimal-integer .

binary-scale ::=
    "S" [ add-operator ] | "s" [ add-operator ] .

character-constant ::=
    [ character-prefix ] character-string
    [ character-suffix ] .

character-count ::= base-integer | prefixed-integer | "*" .

character-data ::
    [ character-prefix ] character-string
    [ character-count ] [ character-suffix ] .

character-prefix ::= "A" | "a" | "C" | "c" | "E" | "e" .

character-string ::= "'" { string-character } "'" .

character-suffix ::=
    "H" | "h" | "L" | "l" | "R" | "r" | "Z" | "z" .

complement-character ::= "#" .

complex register ::=
    complex-register-mnemonic register-designator .

complex-register-mnemonic ::=†
    "A" | "B" | "SB" | "SM" | "SR" | "S" | "T" | "ST" | "V" .

constant ::=
    floating-constant | integer-constant |
    character-constant .
```

---

† All characters within these terminal symbols can be either uppercase or lowercase.

```
data-item ::=
    floating-data | integer-data | character-data .

decimal-digit ::= octal-digit | "8" | "9" .

decimal-exponent ::=
    "E" [ add-operator ] | "e" [ add-operator ] |
    "D" [ add-operator ] | "d" [ add-operator ] .

decimal-fraction ::= "." decimal-integer .

decimal-integer ::= decimal-digit { decimal-digit } .

decimal-prefix ::= "D'" | "d'" .

element ::= special-element | constant | symbol | literal .

element-prefix ::= "P." | "p." | "W." | "w." .

embedded-argument ::= "(" { argument } ")" .

expression ::=
    embedded-argument |
    [ add-operator ] term { add-operator term } .

expression-ending ::=
    expression [ expression-separator
    [ register-or-expression [ suffix ] ] ] .

expression-prefix ::=
    prefixed-expression-character |
    complement-character prefixed-expression-character .

expression-separator ::=
    suffix | logical-character | "=" | .
    [ complement-character ] prefixed-expression-character .

floating-constant ::=
    [ decimal-prefix ] floating-decimal
    [ binary-scale decimal-integer ] ] .

floating-data ::= [ sign ] floating-constant .

floating-decimal ::=
    decimal-integer decimal-fraction
    [ decimal-exponent decimal-integer ] |
    decimal-integer "."
    [ decimal-exponent decimal-integer ] |
    decimal-integer decimal-exponent decimal-integer |
    decimal-fraction [ decimal-exponent decimal-integer ] .
```

```
functional ::= identifier | "=" .

hex-digit ::=
    decimal-digit |
    "A" | "a" | "B" | "b" | "C" | "c" | "D" | "d" |
    "E" | "e" | "F" | "f" .

hex-integer ::= hex-digit { hex-digit } .

hex-prefix ::= "X'" | "x'" .

identifier ::=
    initial-identifier-character
    [ identifier-character [ identifier-character
    [ identifier-character [ identifier-character
    [ identifier-character [ identifier-character
    [ identifier-character ] ] ] ] ] ] .

identifier-character ::=
    initial-identifier-character | decimal-digit .

initial-expression ::=
    [ prefix ] [ expression-prefix ] expression
    [ expression-separator [ register-ending ] ] |
    [ prefix ] [ expression-prefix ] expression
    [ expression-separator [ expression-ending ] ] |
    expression [ expression-separator
    [ register-ending ] ] |
    expression [ expression-separator
    [ expression-ending ] ] .

initial-identifier-character ::= letter | "$" | "%" | "@" .

initial-mnemonic-character ::=
    letter | decimal-digit | "$" | "%" | "&" | "'" | "*" | "+" | "-" |
                ":" | "/" | ":" | "=" | "?" | "\" | "`" | "|" | "~" .

initial-register ::=
    [ prefix ] [ register-prefix ] register
    [ register-separator [ register-ending ] ] |
    [ prefix ] [ register-prefix ] register
    [ register-expression-separator
    [ register-ending ] ] |
    [ prefix ] [ register-prefix ] register
    [ register-expression-separator
    [ expression-ending ] ] |
    [ prefix ] [ register-prefix ] register
    [ special-register-separator [ register-ending ] ] .
```

```
integer-constant ::=
    base-integer [ binary-scale base-integer ] |
    octal-prefix octal-integer
    [ binary-scale octal-integer ] |
    decimal-prefix decimal-integer
    [ binary-scale decimal-integer ] |
    hex-prefix hex-integer [ binary-scale hex-integer ] .

integer-data ::= [ sign ] integer-constant .

letter ::=
    "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
    "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
    "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
    "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
    "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
    "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" .

literal ::= "=" data-item .

logical-character ::= "&" | "!" | "\" .

mnemonic ::=
    initial-mnemonic-character
    [ mnemonic-character [ mnemonic-character
    [ mnemonic-character [ mnemonic-character
    [ mnemonic-character [ mnemonic-character
    [ mnemonic-character ] ] ] ] ] ] .

mnemonic-character ::= initial-mnemonic-character | "@" .

multiply-operator ::= "*" | "/" .

octal-digit ::=
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" .

octal-integer ::= octal-digit { octal-digit } .

octal-prefix ::= "O'" | "o'" .

prefix ::= "(" | "[" .

prefixed-element ::= [ "#" ] [ element-prefix ] element .

prefixed-expression-character ::= "<" | ">" .

prefixed-integer ::=
    octal-prefix octal-integer |
    decimal-prefix decimal-integer |
    hex-prefix hex-integer .
```

```
prefixed-register-character ::=
    "F" | "f" | "H" | "h" | "I" | "i" | "P" | "p" |
    "Q" | "q" | "R" | "r" | "Z" | "z" .

qualified-symbol ::= "/" [ identifier ] "/" identifier .

register ::= complex-register | simple-register .

register-designator ::=
    octal-digit [ octal-digit [ octal-digit [ octal-digit ] ] ] |
    "." integer-constant |
    "." symbol .

register-ending ::=
    register [ register-separator [ register [ suffix ] ] ] |
    register [ register-expression-separator
    [ register-or-expression [ suffix ] ] ] |
    register [ special-register-separator
    [ register [ suffix ] ] ] .

register-expression-separator ::=
    expression-separator | arithmetic-character .

register-or-expression ::= register | expression .

register-prefix ::=
    expression-prefix |
    complement-character [ prefixed-register-character ] |
    arithmetic-character [ prefixed-register-character ] |
    prefixed-register-character .

register-separator ::=
    arithmetic-character prefixed-register-character .

result-field ::= functional | symbolic .

sign ::= "+" | "-" | "#" .

simple-register ::= simple-register-mnemonic .

simple-register-mnemonic ::=†
    "CA" | "CE" | "CI" | "CL" | "MC" | "RT" | "SB" | "SM" | "VL" | "VM" |
    "XA" .
```

† All characters within these terminal symbols can be either uppercase or lowercase.

```
special-element ::=
    "*A" | "*a" | "*B" | "*b" | "*O" | "*o" | "*P" | "*p" | "*W" | "*w" |
    "*" .

special-register-separator ::= complement-character .

string-character ::=
    "'" | " " | "!" | """ | "#" | "$" | "%" | "&" | "(" |
    ")" | "*" | "+" | "," | "-" | "." | "/" | "0" | "1" |
    "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | ":" |
    ";" | "<" | "=" | ">" | "?" | "@" | "A" | "B" | "C" |
    "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" |
    "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
    "V" | "W" | "X" | "Y" | "Z" | "[" | "\" | "]" | "^" |
    "_" | "`" | "a" | "b" | "c" | "d" | "e" | "f" | "g" |
    "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
    "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" |
    "z" | "{" | "|" | "}" | "~" .

subfield ::=
    initial-register | mnemonic | initial-expression .

subfield-separator ::= "," .

suffix ::= ")" | "]" .

symbol ::= unqualified-symbol | qualified-symbol .

symbolic ::=
    [ subfield ] [ subfield-separator [ subfield ]
    [ subfield-separator [ subfield ] ] ] .

term ::=
    prefixed-element { multiply-operator prefixed-element } .

unqualified-symbol ::= identifier .
```

B. PSEUDO INSTRUCTION INDEX

† Available on CRAY X-MP and CRAY 1 Computer Systems only

Name	Definition	Page
LOC	Set * counter	5-32
LOCAL	Specify local symbols	5-152
MACRO	Macro definition	5-104
MICRO	Micro definition	5-86
MICSIZE	Set redefinable symbol to micro size	5-56
MLEVEL	Message priority	5-42
NEXTDUP	Premature exit of the current iteration of a duplication expansion	5-147
OCTMIC	Octal micros	5-89
OPDEF	Operation definition	5-123
OPSYN	Synonymous operation	5-154
ORG	Set *O counter	5-30
QUAL	Qualify symbols	5-13
SECTION	Section assignment	5-18
SET	Set symbol	5-55
SKIP	Unconditionally skip statements	5-79
SPACE	List blank lines	5-48
STACK	Increment the size of the stack	5-29
START	Specify program entry	5-10
STOPDUP	Stop duplication	5-147
SUBTITLE	Specify listing subtitle	5-50
TEXT	Declare beginning of global text source	5-51
TITLE	Specify listing title	5-50
VWD	Variable word definition	5-63

C. LISTING MESSAGES

Listing messages are generated by the assembler when a syntax or semantic error is encountered. Table C-1 lists and briefly describes the listing messages that are generated by CAL.

Table C-1. Listing Messages

Message	Priority	Description
6	Error	Micro was previously defined with a redefinable attribute
7	Error	Micro was previously defined with a nonredefinable attribute
8	Error	Micro was previously defined
10	Error	Assembler module is not terminated properly
11	Error	Maximum allowable address was exceeded in the literal section
12	Error	Definition is not terminated properly
13	Error	Skipping is not terminated properly
14	Warning	Symbol is not defined
15	Warning	Symbol matches the syntax of a register
16	Error	Symbol was previously defined with a redefinable attribute
17	Error	Symbol was previously defined with a nonredefinable attribute
18	Error	Symbol was previously defined
19	Warning	Symbol is immobile or relocatable relative to an empty section

Table C-1. Listing Messages (continued)

Message	Priority	Description
20	Caution	Input line truncation has occurred
21	Caution	Micro character was not terminated
22	Error	Micro character was not terminated after embedded micros were edited
23	Warning	Micro name is undefined and ignored
24	Error	Micro name is undefined after embedded micros were edited
25	Warning	Last line of input file; no appending is done.
26	Warning	Location field was not defined in the formal definition
27	Warning	Field is ignored; a comment was expected.
28	Error	Pseudo instruction is invalidly placed
29	Error	Machine instruction is invalidly placed
35	Warning	Location field is ignored
36	Error	Syntax error; null instruction is not recognized.
37	Error	Syntax error; instruction is not recognized.
38	Error	Field contains too many subfield representations
39	Error	Character is not recognized as a subfield separator
40	Error	Empty result field in prototype statement is not allowed
41	Warning	Functional in this prototype statement has been defined previously

Table C-1. Listing Messages (continued)

Message	Priority	Description
42	Warning	Syntax of this prototype statement has been defined previously
43	Error	Parameter has been used previously within this definition
44	Error	Expression-argument-value syntax is invalid
45	Error	Unrecognized characters are skipped after expression-argument-value
46	Error	Unrecognized characters are skipped after the string-argument-value
47	Error	Keyword parameter name is unknown
48	Warning	Keyword parameter has been used previously within this call
49	Warning	Local parameter has been used previously within this definition
50	Error	Name was expected but not found
51	Error	Name was expected but the end of the statement was encountered
52	Error	Name is terminated illegally
53	Error	Name is terminated illegally by the end of the statement
56	Error	Expression was terminated illegally
57	Error	Functional was expected but not found
58	Error	Functional was terminated illegally
59	Error	Functional was terminated illegally by the end of the statement
60	Error	Embedded-argument is semantically prohibited in an expression

Table C-1. Listing Messages (continued)

Message	Priority	Description
61	Error	Only one external is allowed in an expression
62	Error	External term must never be preceded by a minus operator
63	Warning	Term has parcel attribute but expression has word attribute
64	Warning	Term has parcel attribute but expression has mixture
65	Warning	Term has word attribute but expression has parcel attribute
66	Warning	Term has word attribute but expression has mixture
67	Error	Expression is relative to more than one location
68	Error	Expression cannot be immobile or relocatable and contain an element
69	Error	External element must be the only element within a term
70	Warning	Relocatable attribute changed to absolute; zero width destination
71	Warning	External attribute changed to absolute; zero width destination.
72	Caution	Destination of expression is shorter than relocatable parcel width
73	Caution	Destination of expression is shorter than relocatable word width
74	Warning	Positive expression result is truncated
75	Warning	Negative expression result is truncated

Table C-1. Listing Messages (continued)

Message	Priority	Description
76	Error	More than one immobile or relocatable was found in the term
77	Error	Partial term preceding the division operator must be absolute
78	Error	Element following a division operator must be absolute
79	Error	Division by zero is not allowed
80	Warning	Partial term with parcel address is multiplied by parcel element
81	Warning	Partial term with word address is multiplied by parcel element
82	Warning	Partial term with parcel address is multiplied by word element
83	Warning	Partial term with word address is multiplied by word element
84	Warning	Partial term with value address is divided by parcel element
85	Warning	Partial term with word address is divided by parcel element
86	Warning	Partial term with value address is divided by word element
87	Warning	Partial term with parcel address is divided by word element
88	Error	Prefixed-element was expected, but the end of the statement was encountered
89	Error	Expression element was expected after #
90	Error	Expression element was expected but not found

Table C-1. Listing Messages (continued)

Message	Priority	Description
91	Error	Expression element expected; but end of statement was encountered
92	Error	Special-element is allowed only within an assembler module
93	Warning	Origin counter is not on a parcel boundary
94	Warning	Location counter is not on a parcel boundary
95	Error	Floating-constant cannot be complemented
96	Warning	Immobile attribute change to absolute; zero width destination
97	Error	Numeric base is not decimal, insert a decimal-prefix
98	Error	Floating-constant is too large to evaluate
99	Warning	Double precision floating-constant is converted to single precision
100	Warning	Overflow was detected while evaluating floating-constant
101	Warning	Exponent underflow was detected while evaluating floating-constant
102	Warning	Exponent overflow was detected while evaluating floating-constant
103	Error	Binary-scale is too large to evaluate
104	Warning	Overflow was detected while evaluating binary-scale
105	Caution	Binary-scale value is out of range
106	Warning	Exponent underflow is due to binary-scale value

Table C-1. Listing Messages (continued)

Message	Priority	Description
107	Warning	Exponent overflow is due to binary-scale value
108	Error	Integer-constant is too large to evaluate
109	Error	Integer-constant contains nonoctal digits; the base is octal
110	Warning	Overflow was detected while evaluating floating-integer
111	Error	Binary-scale contains nonoctal digits; the base is octal
112	Warning	ASCII character string is truncated
113	Warning	CDC character string is truncated
114	Warning	EBCDIC character string is truncated
115	Caution	Special-element is not on a word boundary
116	Error	Special-element is not absolute and cannot be complemented
117	Caution	Constant is not on a word boundary
118	Error	Constant is not absolute and cannot be complemented
119	Caution	Symbol is not on a word boundary
120	Error	Symbol is not absolute and cannot be complemented
121	Caution	Literal is not on a word boundary
122	Error	Literal is not absolute and cannot be complemented
123	Error	Data-item of a literal must never have a length of zero

Table C-1. Listing Messages (continued)

Message	Priority	Description
124	Error	Symbol is undefined
125	Error	Unqualified symbol is undefined
126	Error	Qualified symbol is undefined
127	Error	Count symbol must have absolute value attributes
128	Error	Unqualified count symbol must have absolute value attributes
129	Error	Qualified count symbol must have absolute value attributes
130	Error	Expression cannot have a parcel attribute
131	Error	Expression cannot have a word attribute
132	Error	Expression cannot have an absolute attribute
133	Error	Expression cannot have an immobile attribute
134	Error	Expression cannot have a relocatable attribute
135	Error	Expression cannot have an external attribute
136	Error	Expression cannot have a negative value
137	Error	Count was expected but not found
138	Error	Count was expected, but the end of the statement was encountered
139	Error	Count value is out of range; too low
140	Error	Count value is out of range; too high
142	Warning	Integer evaluation of floating-constant has been performed

Table C-1. Listing Messages (continued)

Message	Priority	Description
143	Warning	Floating-constant has been complemented
146	Caution	Expression is relocatable relative to a task common and is nonzero
159	Error	File cannot be opened
160	Error	File can be used only once at a given time
161	Error	Maximum number of allowable sections is exceeded
162	Error	Blank common name is already in use with different attributes
163	Error	Common name is already in use with different attributes
164	Error	Section attribute is not recognized
165	Error	Type attribute can be specified only once
166	Error	Location attribute can be specified only once
167	Error	Task common section must be named
168	Error	Maximum section memory size has been exceeded
170	Warning	Text mode is terminated due to the start of an assembler module
171	Warning	Specified text name has replaced the current text name
172	Caution	Text mode is not currently enabled
173	Error	Unrecognizable attribute; "V", "P", or "W" was expected
174	Warning	Immobile expression cannot be converted to a value
175	Warning	Relocatable expression cannot be converted to a value

Table C-1. Listing Messages (continued)

Message	Priority	Description
176	Error	Entry symbol is undefined
177	Error	Maximum number of allowable entries was exceeded
178	Error	Entry symbol is not redefinable
179	Error	Entry symbol is not relocatable relative to a stack
180	Error	Entry symbol is not relocatable relative to a task common section
181	Error	Entry symbol cannot be external
182	Error	Primary entry was previously defined; only one entry is allowed
183	Error	Primary entry cannot be absolute
184	Error	Primary entry cannot be immobile
185	Error	Primary entry must be relative to either a code or a mixed section
188	Caution	Pseudo was not found within a definition
190	Error	Not supported when targeting for a CRAY-1 Computer System
191	Error	Not supported when targeting for a CRAY X-MP Computer System
192	Error	Not supported when targeting for a CRAY-2 Computer System
196	Error	Unrecognized characters were skipped
197	Error	Final expression value is too large
198	Error	"/" was expected after the count
199	Error	Data-item was expected but not found

Table C-1. Listing Messages (continued)

Message	Priority	Description
200	Warning	Instruction is redefined
201	Error	Instruction is undefined
202	Error	End of statement was not expected after the numeric name
203	Error	Comma was expected after the numeric name
204	Error	End of statement was not expected after the first expression
205	Error	Comma was expected after the first expression of the condition
206	Error	Condition was expected after the comma
207	Error	Comma was expected after the condition
209	Error	Conditional relation was expected but not found
210	Caution	Attributes of the two expressions do not match
214	Error	End of statement was not expected after the first string
215	Error	Comma was expected after the first string of the condition
218	Error	Invalid target machine characteristic is specified
220	Error	Attribute was expected, but the end of the statement was encountered
221	Error	Attribute was expected
222	Error	Comma was expected after the attribute
223	Error	Expression was expected after the comma

Table C-1. Listing Messages (continued)

Message	Priority	Description
228	Warning	Skip count exists, but the skip name is used instead
231	Error	Message priority is not recognized
232	Comment	User-defined message
233	Note	User-defined message
234	Caution	User-defined message
235	Warning	User-defined message
236	Error	User-defined message
238	Caution	Corresponding expansion was not found
240	Warning	Duplication name exists, but the duplication count is used instead
244	Error	List of arguments was not terminated properly
250	Error	Invalid fill specification
251	Warning	Micro string is truncated due to character count
252	Error	Value of expression specifying the character count is too large
253	Error	Micro name has not been previously defined
254	Error	Invalid case specification
260	Warning	Alignment is relative to task common section
261	Warning	Alignment is relative to stack section
262	Warning	Alignment is relative to Local Memory section
264	Caution	A comment has been previously specified

Table C-1. Listing Messages (continued)

Message	Priority	Description
270	Caution	Location field is ignored by this pseudo
271	Caution	Operand field is ignored by this pseudo
272	Warning	Location field was expected by this pseudo but was not found
273	Error	Location field is required by this pseudo but was not found
274	Warning	Operand field was expected by this pseudo but was not found
275	Error	Operand field is required by this pseudo but was not found
276	Warning	Matching delimiter character to terminate the character string was not found
277	Warning	String is too long; the string is truncated
280	Caution	Stack is currently empty
281	Caution	Current stack entry is not redefinable
282	Warning	Invalid pseudo option is specified
283	Caution	Listing cannot be disabled
284	Caution	Option is not redefinable
285	Caution	Option is repeated
286	Warning	Option is ambiguous because of a previous option in the list
287	Caution	List pseudo is not processed
290	Error	Maximum number of allowable externals was exceeded
291	Error	External attribute is not recognized

Table C-1. Listing Messages (continued)

Message	Priority	Description
292	Caution	External attribute has been previously specified
293	Error	External address attribute has been previously specified
294	Error	External linkage attribute has been previously specified
295	Caution	Soft externals are not supported within the operating system
300	Error	Expression must be "0"
301	Error	Expression must be "1"
302	Error	Expression must be either "0" or "1"
303	Error	Relocatable expression must be relative to Common Memory
304	Error	Relocatable expression must be relative to Local Memory
305	Error	Relocatable expression must be relative to a mixed or code section
310	Error	Register designator value is too large
311	Error	Register designator must have a value attribute
312	Error	Register designator must not have an immobile attribute
313	Error	Register designator must not have a relocatable attribute
314	Error	Register designator must not be an external
315	Error	Register designator value does not fit into a 3-bit opcode field

Table C-1. Listing Messages (continued)

Message	Priority	Description
316	Error	Register designator value does not fit into a 6-bit opcode field
320	Error	Register designator does not equal that of the first register
321	Error	Register designator does not equal that of the second register
322	Error	Register designator must not equal zero
323	Warning	Register designator value is zero, possible instruction error
324	Error	Register designator value must not be greater than octal 37
325	Error	Register designator value must be zero
326	Warning	Vector recursion; designator equals that of the first register
327	Comment	Vector recursion; designator equals that of the first register
328	Warning	Zero vector; designator equals that of the first register
340	Warning	Expression has a relative attribut of immobile
341	Warning	Expression is relocatable relative to a task common section

D. DIAGNOSTIC MESSAGES

CAL generates five levels of diagnostic messages that are divided into two classes: user information about the assembly (comment, note, and caution) and CAL assembler messages (warnings and errors).

Priority	Description
Comment	Comment priority messages are numbered in the range from 1 to 99 and provide general assembler introductions and statistics.
Note	Note priority messages are numbered in the range from 1 to 99 and provide information such as the number or caution and warning messages found while assembling.
Caution	Caution priority messages are numbered in the range from 1 to 99 and indicate user errors.
Warning	Warning priority messages are numbered 100 and greater and indicate incompatibility between different binary load tables; assembly continues.
Error	Error priority messages are numbered 100 and greater and report diagnostic messages that were detected by the assembler. After the message is issued, assembly is aborted.

If you receive a diagnostic message with a priority of warning or error, contact your local site analyst.

CAL diagnostic messages are always prefixed by the operating system as follows:

    ca**nnn** - [CAL]

where **nnn** is the number of the diagnostic message. For example:

    ca001 - [CAL] CAL Version 2 - Release (**month/day/year**)

Warning and error messages are printed only if the assembler is malfunctioning. Since this condition is rare, these messages are not listed in this manual. For more information about diagnostic messages, see section 2, Operating Systems.

Table D-1 lists and briefly describes diagnostic messages 1 through 99.

Table D-1. Diagnostic Messages

Message	Priority	Description
1	Comment	CAL Version 2 - Release *version* (*mm/dd/yy*)
2	Comment	Total number of segments processed: *n*
3	Comment	Total number of source lines read: *n*
4	Comment	Total number of statements processed: *n*
5	Comment	Total time of assembly: *n* seconds
20	Caution	'*option name*' is not a valid *type* option
21	Note	'*parameter*' invocation statement parameter is no longer supported
22	Caution	Primary CPU option '*option name*' is not recognized; the default is used.
23	Caution	Conflicting *trait* CPU characteristic '*charac*' specified
24	Caution	'*trait*' expected to follow CPU characteristic '*charac*'
25	Caution	Input file '*filename*' was repeated on the invocation statement
26	Caution	Attempt to open file '*filename*' failed
27	Caution	Valid options '*options*'
28	Caution	The '*option name*' option has too many arguments
29	Caution	The '*option name*' option has been repeated
30	Caution	File '*filename*' has been specified more than once
31	Caution	An input file was required on the invocation statement

Table D-1. Diagnostic Messages (continued)

Message	Priority	Description
32	Caution	Only one input file is allowed on the invocation statement
33	Caution	The '*option name*' option contradicts a prvevious '*option name*' option
34	Caution	'*option name*' is not a valid *option* name
45	Comment	Segment '*name*' -- *user message*
50	Comment	Segment '*name*' has *n* comment messages
51	Comment	Segment '*name*' has *n* note messages
52	Note	Segment '*name*' has *n* caution messages
53	Note	Segment '*name*' has *n* warning messages
54	Caution	Segment '*name*' has *n* error messages
56	Note	Message count was exceeded, *n* messages are listed
70	Caution	Bad binary definition file '*file*'
71	Caution	Incompatible version of binary definition file '*file*'
72	Caution	Binary definition file '*file*' was not built for this machine type
74	Note	Symbol '*name*' is redefined in file '*file*'
75	Note	Micro '*name*' is redefined in file *file*
76	Note	Macro '*name*' in file '*file*' replaces previous definition
77	Note	Opsyn '*name*' in file '*file*' replaces previous definition
78	Note	Opdef '*name*' in file '*file*' replaces previous definition
90	Caution	Assembly errors were encountered

E. CHARACTER SET

Character	ASCII Code (Octal/Hex)	EBCDIC Code (Hex)	CDC Code (Octal)
NUL	000/00	00	None
SOH	001/01	01	None
STX	002/02	02	None
ETX	003/03	03	None
EOT	004/04	37	None
ENQ	005/05	2D	None
ACK	006/06	2E	None
BEL	007/07	2F	None
BS	010/08	16	None
HT	011/09	05	None
LF	012/0A	25	None
VT	013/0B	0B	None
FF	014/0C	0C	None
CR	015/0D	0D	None
S0	016/0E	0E	None
SI	017/0F	0F	None
DLE	020/10	10	None
DC1	021/11	11	None
DC2	022/12	12	None
DC3	023/13	13	None
DC4	024/14	3C	None
NAK	025/15	3D	None
SYN	026/16	32	None
ETB	027/17	26	None

Character	ASCII Code (Octal/Hex)	EBCDIC Code (Hex)	CDC Code (Octal)
CAN	030/18	18	None
EM	031/19	19	None
SUB	032/1A	3F	None
ESC	033/1B	27	None
FS	034/1C	1C	None
GS	035/1D	1D	None
RS	036/1E	1E	None
US	037/1F	1F	None
Space	040/20	40	55
!	041/21	5A	66
"	042/22	7F	64
#	043/23	7B	60
$	044/24	5B	53
%	045/25	6C	63
&	046/26	50	67
'	047/27	7D	70
(	050/28	4D	51
)	051/29	5D	52
*	052/2A	5C	47
+	053/2B	4E	45
,	054/2C	6B	56
-	055/2D	60	46
.	056/2E	4B	57
/	057/2F	61	50
0	060/30	F0	33
1	061/31	F1	34
2	062/32	F2	35
3	063/33	F3	36
4	064/34	F4	37

Character	ASCII Code (Octal/Hex)	EBCDIC Code (Hex)	CDC Code (Octal)
5	065/35	F5	40
6	066/36	F6	41
7	067/37	F7	42
8	070/38	F8	43
9	071/39	F9	44
:	072/3A	7A	00
;	073/3B	5E	77
<	074/3C	4C	72
=	075/3D	7E	54
>	076/3E	6E	73
?	077/3F	6F	71
@	100/40	7C	74
A	101/41	C1	01
B	102/42	C2	02
C	103/43	C3	03
D	104/44	C4	04
E	105/45	C5	05
F	106/46	C6	06
G	107/47	C7	07
H	110/48	C8	10
I	111/49	C9	11
J	112/4A	D1	12
K	113/4B	D2	13
L	114/4C	D3	14
M	115/4D	D4	15
N	116/4E	D5	16
O	117/4F	D6	17
P	120/50	D7	20
Q	121/51	D8	21

Character	ASCII Code (Octal/Hex)	EBCDIC Code (Hex)	CDC Code (Octal)
R	122/52	D9	22
S	123/53	E2	23
T	124/54	E3	24
U	125/55	E4	25
V	126/56	E5	26
W	127/57	E6	27
X	130/58	E7	30
Y	131/59	E8	31
Z	132/5A	E9	32
[	133/5B	AD	61
\	134/5C	E0	75
]	135/5D	BD	62
^	136/5E	5F	76
_	137/5F	6D	65
'	140/60	79	None
a	141/61	81	None
b	142/62	82	None
c	143/63	83	None
d	144/64	84	None
e	145/65	85	None
f	146/66	86	None
g	147/67	87	None
h	150/68	88	None
i	151/69	89	None
j	152/6A	91	None
k	153/6B	92	None
l	154/6C	93	None
m	155/6D	94	None
n	156/6E	95	None

Character	ASCII Code (Octal/Hex)	EBCDIC Code (Hex)	CDC Code (Octal)
o	157/6F	96	None
p	160/70	97	None
q	161/71	98	None
r	162/72	99	None
s	163/73	A2	None
t	164/74	A3	None
u	165/75	A4	None
v	166/76	A5	None
w	167/77	A6	None
x	170/78	A7	None
y	171/79	A8	None
z	172/7A	A9	None
{	173/7B	C0	None
}	174/7C	6A	None
¦	175/7D	D0	None
~	176/7E	A1	None
DEL	177/7F	07	None

INDEX

INDEX

READER COMMENT FORM

CAL Assembler Version 2 Reference Manual SR-2003

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

DATE _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

Attention:
PUBLICATIONS

READER COMMENT FORM

CAL Assembler Version 2 Reference Manual SR-2003

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

DATE _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

Attention:
PUBLICATIONS