Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual

SR–3108 9.1

Cray Research, Inc.

Copyright © 1992, 1995 Cray Research, Inc. All Rights Reserved. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Cray Research, Inc.

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

Autotasking, CF77, CRAY, Cray Ada, CraySoft, CRAY Y-MP, CRAY-1, HSX, MPP Apprentice, SSD, SUPERCLUSTER, SUPERSERVER, UniChem, UNICOS, and X-MP EA are federally registered trademarks and Because no workstation is an island, CCI, CCMT, CF90, CFT, CFT2, CFT77, ConCurrent Maintenance Tools, COS, Cray Animation Theater, CRAY APP, CRAY C90, CRAY C90D, Cray C++ Compiling System, CrayDoc, CRAY EL, CRAY J90, Cray NQS, Cray/REELlibrarian, CRAY S-MP, CRAY SUPERSERVER 6400, CRAY T90, CRAY T3D, CrayTutor, CRAY X-MP, CRAY XMS, CRAY-2, CRInform, CRI/*Turbo*Kiva, CS6400, CSIM, CVT, Delivering the power . . ., DGauss, Docview, EMDS, HEXAR, IOS, LibSci, ND Series Network Disk Array, Network Queuing Environment, Network Queuing Tools, OLNET, RQS, SEGLDR, SMARTE, SUPERLINK, System Maintenance and Remote Testing Environment, Trusted UNICOS, and UNICOS MAX are trademarks of Cray Research, Inc.

CDC is a trademark of Control Data Systems, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. X Window System is a trademark of the X Consortium, Inc.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

New Features

Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual SR–3108 9.1

This rewrite of the *Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual*, publication SR–3108, supports the 9.1 release of CAL. The following additions and modifications are included:

- Information throughout the manual to support IEEE floating-point hardware on CRAY T90 systems
- New instructions for CRAY T90 systems in appendix E
- Pertinent information from appendix F has been incorporated into the body of the manual and appendix F has been eliminated

Record of Revision

The date of printing or software version number is indicated in the footer. Changes in rewrites are noted by revision bars along the margin of the page.

Version	***Description***
7.0	September 1992. Original printing.
8.3	January 1995. Updated only online. Revised to support the CAL2 8.3 release.
9.0	August 1995. Rewrite to support the CAL 9.0 release.
9.1	November 1995. Updated to support the CAL 9.1 release for IEEE floating-point information for CRAY T90 systems.

Preface

Cray Assembly Language (CAL) symbolically expresses all hardware functions of Cray Research parallel vector processing (PVP) systems. This detailed and precise level of programming is especially helpful for tailoring programs to the architecture of a Cray PVP system and for writing programs that require code that is optimized to the hardware.

In addition to the instruction set, CAL also includes a versatile set of pseudo instructions that provides a variety of options for generating macro instructions, controlling listing output, organizing programs, and so on.

With the release of CAL 9.1, IEEE floating-point hardware is supported on CRAY T90 systems.

Related publications

This manual is written for experienced programmers. The following documents contain additional information that may be helpful:

- *Cray Assembly Language (CAL) for Cray PVP Systems Ready Reference*, publication SQ–3110
- *Symbolic Machine Instructions Reference Manual*, publication SR–0085
- *UNICOS User Commands Reference Manual*, publication SR–2011
- *UNICOS Macros and Opdefs Reference Manual*, publication SR–2403
- *CF90 Commands and Directives Reference Manual*, publication SR–3901
- *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902

- *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903
- *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software manuals that are available to customers.

Ordering publications

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software documents that are available to customers. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, either call the Distribution Center in Mendota Heights, Minnesota, at +1–612–683–5907 or send a facsimile of your request to fax number +1–612–452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers: 1 User commands 1B User commands ported from BSD 2 System calls 3 Library routines, macros, and opdefs 4 Devices (special files) 4P Protocols 5 File formats 7 Miscellaneous topics 7D DWB-related information 8 Administrator commands Some internal routines (for example, the `ddcntl`() routine) do not have man pages associated with them.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
. . .	A vertical ellipsis indicate a continued sequence of code in the sample program.
[ ]	Brackets enclose optional portions of a command line.

The following machine naming conventions may be used throughout this document:

Term	Definition
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems, including the following: CRAY C90 series (CRAY C916, CRAY C92A, CRAY C94, CRAY C94A, and CRAY C98 systems) CRAY C90D series (CRAY C92AD, CRAY C94D, and CRAY C98D systems) CRAY J90 series (CRAY J916 and CRAY J932 systems) CRAY T90 series (CRAY T94, CRAY T916, and CRAY T932 systems)
All Cray Research systems	All configurations of Cray PVP and Cray MPP systems that support this release

The default shell in the UNICOS operating system, referred to in Cray Research documentation as the standard shell, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Portability Guide, Issue 4 (XPG4)

The UNICOS operating system also supports the optional use of the C shell.

Cray UNICOS Version 9.0 is an X/Open Base 95 branded product.

The POSIX standard uses *utilities* to refer to executable programs that Cray Research documentation usually refers to as *commands*. Both terms may appear in this document.

Online information

The following types of online information products are available to Cray Research customers:

- CrayDoc online documentation reader, which lets you see the text and graphics of a document online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Man pages, which describe a particular element of the UNICOS operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.
- UNICOS message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.
- Cray Research online glossary, which explains the terms used in a document. To get a definition, use the `define`(1) command.
- `xhelp` help facility. This online help system is available within tools such as the Program Browser (`xbrowse`) and the MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!publications`

- Send us electronic mail from any system connected to the Internet, using the following Internet address:

  `publications@timbuk.cray.com`

- Contact your Cray Research representative and ask that a Software Problem Report (SPR) be filed. Use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.
- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600
- Send a facsimile of your comments to the attention of “Software Publications Group” in Eagan, Minnesota, at fax number +1–612–683–5599.
- Use the postage-paid Reader’s Comment Form at the back of the printed document.

We value your comments and will respond to them promptly.

Contents

	Page
Preface	iii
Related publications	iii
Ordering publications	iv
Conventions	v
Online information	vii
Reader comments	vii
Introduction [1]	1
Manual organization	1
New features and modifications	2
Capabilities	2
Execution of the CAL assembler	3
Source statement format	3
Assembler listings format	4
Source statement listing format	5
Cross-reference listing format	7
UNICOS System Information [2]	11
as(1) – CAL command line	12
Interactive assembly	19
The UNICOS environment	20
LPP environment variable	20
TMPDIR shell variable	22
MSG_FORMAT error message format	24
TARGET shell variable	24
Binary definition files	25
Defining a binary definition file	26
Symbols	27
Macros	27
Opdefs	27
Opsyns	27
Micros	28
Creating binary definition files	29
Using binary definitions files	30
CPU compatibility checking	30
Multiple references to a definition	30
Symbols	30
Macros	31
Opdefs	31
Opsyn	31
Micros	32
The CAL Program [3]	33
Program segment	33
Program module	33
Global definitions	35
Source statement	36
New format	37
Location field	38
Result field	38
Operand field	39
Comment field	39
Old format	40
Location field	40
Result field	41
Operand field	41
Comment field	41
Statement editing	41
Micro substitution	43
Append	44

	Page
Concatenate	44
Continuation	44
Comment	44
Actual statements and edited statements	45
Instructions	46
Assembler-defined instructions	46
Pseudo instructions	47
Machine instructions	47
User-defined instructions	47
Micros	48
Sections	53
Local sections	54
Main sections	54
Literals section	54
Sections defined by the SECTION pseudo instruction	55
Common sections	55
Section stack buffer	56
Origin counter	58
Location counter	58
Word-bit-position counter	58
Force word boundary	59
Parcel-bit-position counter	59
Force parcel boundary	59
Cray Assembly Language [4]	61
Register designators	61
Vector length register (VL)	65
Vector mask register (VM)	66
Names	67
Symbols	69
Symbol qualification	70
Unqualified symbol	70
Qualified symbols	71
Symbol definition	72
Symbol attributes	73
Address attributes	73
Relative attributes	74
Redefinable attributes	75
Symbol reference	75
Data	76
Constants	76
Floating constant	76
Integer constant	79
Character constants	81
Data items	82
Floating data item	82
Integer data item	83
Character data item	84
Literals	85
Special elements	89
Element prefixes	90
Parcel-address prefix	91
Word-address prefix	93
Expressions	94
Add-operator	94
Terms	95
Prefixed-elements	96
Multiply-operator	97
Term attributes	97
Expression evaluation	100
Evaluating immobile and relocatable terms with coefficients	104
Expression attributes	110
Relative attributes	111
Address attributes	112
Truncating expression values	112
Pseudo Instructions [5]	117
Program control	119

	Page
Loader linkage	119
Mode control	119
Section control	120
Message control	121
Listing control	121
Symbol definition	122
Data definition	122
Conditional assembly	123
Micro definition	124
File control	126
Defined Sequences [6]	127
Similarities among defined sequences	128
Editing	128
Definition format	129
Formal parameters	130
Instruction calls	132
Interaction with the INCLUDE pseudo instruction	134
Macros (MACRO)	134
Macro definition	136
Macro calls	142
Operation definitions (OPDEF)	154
Opdef definition	160
Opdef calls	166
Duplication (DUP)	171
Duplicate with varying argument (ECHO)	174
Ending a macro or operation definition (ENDM)	177
Premature exit from a macro expansion (EXITM)	177
Ending duplicated code (ENDDUP)	178
Premature exit of the current iteration of duplication expansion (NEXTDUP)	179
Stopping duplication (STOPDUP)	180
Specifying local unique character string replacements (LOCAL)	184
Synonymous operations (OPSYN)	186
Pseudo Instruction Descriptions [A]	189
User Messages [B]	281
Character Set [C]	353
Symbolic Instruction Summary [D]	359
Instructions [E]	369
Common instructions	370
CRAY J90 and CRAY Y-MP specific instructions	384
CRAY C90 specific instructions	386
CRAY J90 specific instructions	389
CRAY T90 specific instructions	390
Bit Matrix multiply instructions	397
Special register values and logical operators	399
Index	401

Figures

Figure 1.	Page header format	4
Figure 2.	Source statement listing format	5
Figure 3.	Cross-reference listing format	7
Figure 4.	CAL program structure	28
Figure 5.	CAL program organization	34
Figure 6.	ASCII character with left-justification and blank-fill	87
Figure 7.	ASCII character with left-justification and zero-fill	87

		Page
Figure 8.	ASCII character with right-justification and zero-fill	88
Figure 9.	ASCII character with right-justification in 8 bits	88
Figure 10.	Word/parcel conversion for 6 words	92
Figure 11.	Diagram of an expression	94
Figure 12.	Diagram of a term	94
Figure 13.	Address attribute assignment chart	100
Figure 14.	64-bit binary representation of ASCII abc, left-justified	102
Figure 15.	64-bit binary representation of 1	102
Figure 16.	Binary representation of ASCII abc, right-justified in 9 bits	102
Figure 17.	Result of VWD with 9-bit destination field	103
Figure 18.	64-bit binary representation of the complement of 1	103
Figure 19.	64-bit binary representation of 1	103
Figure 20.	Binary representation of the complement of 1 stored in the rightmost bits of a 4-bit field	104
Figure 21.	Result of VWD with 4-bit destination field	104
Figure 22.	64-bit binary representation of –1	113
Figure 23.	Truncated value of –1 stored in a 5-bit field	113
Figure 24.	64-bit binary representation of 5	113
Figure 25.	Truncated value of 5 stored in a 3-bit field	113
Figure 26.	64-bit binary representation of 5	114
Figure 27.	Truncated value of 5 stored in a 2-bit field	114
Figure 28.	BITP example – zoning parcel A	195
Figure 29.	BITP example – parcel b set by VWD instruction	195
Figure 30.	BITP example – resetting the pointer	195
Figure 31.	BITP example – result of a BITP followed by a VWD	195
Figure 32.	Storage of unlabeled data items	207
Figure 33.	Storage of labeled and unlabeled data items	208
Figure 34.	Storage of CDC character data item	208

Tables

		Page
Table 1.	Logical and numeric traits	15
Table 2.	List options	17
Table 3.	Register designations	62
Table 4.	Character set	353
Table 5.	Register entry instructions	359
Table 6.	Interregister transfers	360
Table 7.	Memory transfers	361
Table 8.	Program jumps and exits	362
Table 9.	Bit count instructions	362
Table 10.	Shift instructions	363
Table 11.	Integer arithmetic operations	363
Table 12.	Floating-point operations	364
Table 13.	Logical operations	365
Table 14.	Bit matrix multiply instructions	366
Table 15.	Pass and breakpoint instructions	366
Table 16.	Monitor operations	366
Table 17.	Common symbolic machine instructions	370
Table 18.	CRAY J90 and CRAY Y-MP symbolic machine instructions	384
Table 19.	CRAY C90 symbolic machine instructions	386
Table 20.	CRAY J90 symbolic machine instructions	389
Table 21.	CRAY T90 symbolic machine instructions	390
Table 22.	Bit matrix multiply symbolic machine instructions	397
Table 23.	Special register values and logical operators	399

Introduction [1]

Cray Assembly Language (CAL) is a powerful symbolic language that generates object code for execution on Cray PVP systems.

Two types of CPUs are available with CRAY T90 systems. The first type of CPU uses the same type of floating-point format as all other Cray PVP systems. The second type of CPU conforms with the Institute of Electrical and Electronics Engineers (IEEE) standard 754, and except for minor differences, supports all 64-bit numeric representations, arithmetic operations, rounding modes, and exception handling.

CAL is supported on all Cray PVP systems with somewhat different instruction sets for each product line. The instruction sets for each machine are presented in appendix E, page 369.

Manual organization
1.1

This publication is organized as follows:

Section	Description
2	Describes the CAL invocation statement that executes under the UNICOS operating system. Section 2 also describes binary definition files.
3	Describes the organization of a CAL program.
4	Describes the statement syntax of the CAL program.
5	Describes the use of pseudo instructions.
6	Describes defined sequences available within CAL.
A	Lists descriptions of CAL pseudo instructions in alphabetical order.
B	Lists all CAL user messages.
C	Lists the character sets that CAL supports.

Section	Description
D	Provides tables of the symbolic machine instructions for Cray PVP systems.
E	Provides a table of all symbolic machine instructions for Cray PVP systems. The instructions are listed in numeric order by the opcode and a brief description of the instruction is given.

New features and modifications
1.2

The new instructions necessary to support IEEE floating-point format on CRAY T90 systems can be found in appendix E, page 369.

Capabilities
1.3

CAL provides the following capabilities:

- The free-field source statement format of CAL lets you control the size and location of source statement fields.
- With some exceptions, you can enter source statements in uppercase, lowercase, or mixed-case letters.
- You can assign code or data segments to specific areas to control local and common sections.
- You can use preloaded data by defining data areas during assembly and loading them with the program.
- You can designate data in integer, floating-point, and character code notation.
- You can specify addresses as either word or parcel addresses.
- You can control the content of the assembler listing.
- You can define a character string in a program and substitute the string for each occurrence of its micro name in the program by using micro coding.
- You can define sequences of code in a program or in a library and substitute the sequence for each occurrence of its macro name in the program by using macro coding.

Execution of the CAL assembler
1.4

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

When you specify the CAL invocation statement, the assembler is loaded and begins executing. The parameters used on the invocation statement specify the characteristics of an assembler run, such as the file containing source statements or listing output. For descriptions of the CAL command line, see section 2, page 11, or the `as`(1) man page.

The file containing source statements can include more than one CAL program segment. Each program segment is assembled as it is encountered in the source code. The CAL assembler makes two passes for each program segment. During the first pass, each source language instruction is read, sequences (such as macro instructions) are expanded, machine function codes are generated, and memory is assigned. During the second pass, values are substituted for symbolic operands and addresses; object code and an associated listing are generated.

The object code must be linked and loaded prior to execution to resolve references to external symbols. The absolute file that is created by the link and load process is suitable for execution.

Source statement format
1.5

CAL source programs consist of sequences of source statements. The source statement can be a symbolic machine instruction, pseudo instruction, macro instruction, or opdef instruction. The symbolic machine instructions provide a way of symbolically expressing all functions of a Cray PVP system. Pseudo instructions control the assembly process. Macros and opdefs define instruction sequences that can be called later in a program.

CAL source statements are free-format and can contain any or all of the following fields:

- Location
- Result
- Operand
- Comment

The content of each field is dependant upon the format specified by the `-f` (new format) or `-F` (old format) parameters on the `as` command line or by using the `FORMAT` pseudo instruction. Generally, fields are separated by white space (blanks or tabs). See subsection 3.2, page 36, for more information on the format of source statements.

The following is an example of a CAL source statement:

```
ABC          Si    Sj+Sk        ; Sum
```

In the preceding example `ABC` resides in the location field, `S`*i* in the result field, `S`*j*`+S`*k* in the operand filed, and `;Sum` in the comment field.

Assembler listings format
1.6

CAL generates a source statement listing and a cross-reference listing. You can control the format of these listings by using the listing control pseudo instructions (see subsection 5.6, page 121) or by using the `-i`, `-I`, `-l`, `-L`, `-a`, `-n`, `-h`, and `-H` options on the `as` command line (see subsection 2.1, page 12).

Each page of listing output produced by the CAL assembler contains three header lines. Figure 1 shows the format of the page header.

CAL version #	Title	Global page #
Date and time	Subtitle	Local page #
Section and qualifier	Scale (1–72 characters wide)	Cray Research system

Figure 1. Page header format

The three lines of the page header are described as follows:

- The first line contains, from left to right, the version number of CAL, the title of the program, and a page number that is global over all programs assembled by the current assembly. If you do not specify a title by using a `TITLE` pseudo instruction, the title is taken from the operand field of the `IDENT` pseudo instruction.
- The second line contains, from left to right, the date and time of assembly, a subtitle if specified with a `SUBTITLE` pseudo instruction, and a page number that is local for this listing.
- On the third line, the leftmost entry is a local section name if specified in a `SECTION` pseudo instruction. To the right of the local section name is a symbol qualifier name if specified by a `QUAL` pseudo instruction. The next field is a horizontal scale that is 72 characters wide, numbered from 1 through 72. This scale appears directly over your source code and helps you to differentiate the four fields of your source statements. On the far right of the third line is the name of the Cray Research system for which the code was generated.

Source statement listing format
1.6.1

The format of the source statement listing, as shown in Figure 2, appears directly under the page header and contains five columns of information, as follows:

Location address	Octal code	Line number	Source line or error code	Sequence field

Figure 2. Source statement listing format

The five columns of the source statement listing are described as follows:

- The *location address* column contains the address of the current statement at assembly.

  If the statement is a machine instruction, it lists the parcel address with the parcel identifier a, b, c, or d appended to the word address. Parcels are lettered from left to right within a word. If the statement is not a machine instruction the address is listed as a word address.

- The *octal code* column contains the octal representation of the current instruction or numeric value.

  If the numeric value is an address, the octal code has one of the following suffixes:

  – + (Relocation in program block)
  – C (Common section)
  – D (Dynamic section)
  – S (Stack section)
  – T (Task common)
  – Z (Zero common)
  – : (Immobile attribute)
  – X (External symbol)
  – None (Absolute address)

  The results of several pseudo instructions can also appear in the octal code column:

  – The octal value of symbols defined by the SET, MICSIZE, or = pseudo instruction

  – The octal value of the number of words reserved by the BSS or BSSZ pseudo instruction

  – The octal value of the number of full parcels skipped as a result of the ALIGN pseudo instruction

  – The octal value of the number of characters in a micro string defined by a MICRO, OCTMIC, or DECMIC pseudo instruction

- The *line number* column contains the line number of the source code.

- The *source line* column is 72 characters wide and holds columns 1 through 72 of each source line.
- The *error code* column contains an error message immediately following a statement that contains an error. Error codes are described in appendix B, page 281.
- The *sequence field* column contains either an identifier or information taken from columns 73 through 90 of the source line image. It contains an identifier if the line is an expansion of a macro or opdef, or if the line was edited.

Cross-reference listing format
1.6.2

The assembler generates a cross-reference listing in the format shown in Figure 3. The assembler lists symbols alphabetically and groups them by qualifier if the `QUAL` pseudo instruction has declared qualifiers. If qualifiers were declared, each new qualifier appears on a fresh page. The qualifier name appears on the third line of the page header.

The cross-reference listing does not include unreferenced symbols that are defined between `TEXT` and `ENDTEXT` pseudo instructions and it does not include symbols of the form `%%`*xxxxxx*; x is zero or more identifier characters.

> **Note:** The page header is nearly identical to the page header of the assembler listing; the difference is that the string *Symbol cross reference* is printed out in the middle field of the third line of the cross-reference listing.

CAL version #			Title	Global page #
Date and time			Subtitle	Local page #
Section and qualifier			"Symbol cross reference"	Cray Research system
Symbol	Value	.	Symbol references	

Figure 3. Cross-reference listing format

The information in each column is described as follows:

- The *symbol* column contains the symbol name.
- The *value* column contains the octal value of the symbol.

  A symbol with a parcel address attribute has `a`, `b`, `c`, or `d` appended to the word address. Parcels go from left to right within a word. The octal value of the symbol may have one of the following suffixes:

  - `+` (Relocation in program block)
  - `C` (Common section)
  - `D` (Dynamic section)
  - `S` (Stack section)
  - `T` (Task common)
  - `Z` (Zero common)
  - `:` (Immobile attribute)
  - `X` (External symbol)
  - `None` (Absolute address)
- The *period (.)* column separates the value reference field from the symbol reference fields and is called the separator.
- The *symbol references* column contains one or more references to the symbol.

  The assembler references symbols in the following format:

  page : *line x*

  page is the decimal representation of the local page number in the listing that contains the current reference. The local page number appears in parentheses at the far right end of the second line of the header.

  line is the decimal representation of the line number that contains the reference.

  x represents the type of reference as follows:

  - A *blank* column means the symbol value is used at this point.
  - `D` means the symbol is defined in the location field of an instruction or else by a `SET`, `=`, or `EXT` pseudo instruction.

- E means the symbol is an entry name.
- F means the symbol is used in an expression on an IFE, IFA, or ERRIF conditional pseudo instruction.
- R means the symbol is used in an address expression in a memory read instruction or as a B or T register symbol in an instruction that reads the B or T register.
- S means the symbol is used in an address expression in a memory store instruction or as a B or T register symbol in an instruction that stores a new value in the B or T register.

If a symbol is defined in text between TEXT and ENDTEXT pseudo instructions, the cross-reference listing reports the associated TEXT name on the line below the symbol reference.

If a symbol is defined in a binary definition file, the cross-reference listing reports the associated file name on the line below the symbol reference.

UNICOS System Information [2]

CAL, the Cray assembler, supports binary definition files and interfaces with the UNICOS operating system.

A typical interactive session involves assembling a CAL source file to create a relocatable object file. A link editor or loader processes the object file to create an executable file. This process is accomplished by entering a series of commands at the command line. (For more information, see subsection 2.2, page 19.)

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

CAL generates listing and diagnostic messages during assembly. When the `-l` and `-L` options are specified on the `as`(1) command line and a syntax or semantic error is encountered, the assembler generates listing messages. A message is printed in the listing after each source statement flagged by the assembler and a pointer identifies the location within the source statement that corresponds to the message. The message also is issued to the standard error file.

CAL generates diagnostic messages that provide user information about the assembly (comment, note, and caution) and CAL assembler errors (warning and error). Diagnostic messages are classified by level of severity from low to high, as follows:

- User information about the assembly
  - Comment (statistical information)
  - Note (possible assembly problems)
  - Caution (definite user errors during assembly)
- CAL assembler errors
  - Warning (possible error such as truncation of a value)
  - Error (fatal assembly error, you should check the message and source code carefully for possible mistakes)

Note: To print comment-, note-, and caution-level diagnostic messages to the standard error file, you must specify the `-m` option on the `as`(1) command line.

as(1) – CAL command line
2.1

The UNICOS `as`(1) command invokes the CAL assembler. The format of the `as`(1) command is as follows:

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

The `as`(1) command assembles the specified file. The following options, each a separate argument, can appear in any order, but they must precede the *file* argument:

`-o` *objfile* Relocatable assembly output is 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 the `.o` suffix. A link editor or loader must process *objfile*.

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

`-L` *msgfile* Assembly output source message listing is 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 specified in *bdflist* can be designated using one of the following forms:

- List of files separated by a comma
- List of files enclosed in double quotation marks and separated by a comma and/or one or more spaces

The files listed in *bdflist* are read in the order specified. By default, the binary assembler definitions found in file `/lib/asdef` are also read unless suppressed with the `-B` option.

`-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.

`-D` *micdef* Defines a globally defined constant micro *mname*, as follows:

micdef `::=` *mname*[=[*string*]]

mname must be a valid identifier. If the `=` character is specified, it must immediately follow *mname*. The *string* that immediately follows the `=` character, if any, is associated with *mname*. If you do not specify *string*, *mname* will be associated with an empty string.

If *mname* was defined as a micro by use of a binary definition file, the *mname* specified on the command line overrides the *mname* defined within the binary definition file; in that case, CAL issues a note-level diagnostic message.

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

If you specify the same file for both the `-o` and `-g` options, and the last assembler segment does not contain a module (that is, it contains only the global part of the segment), CAL will not generate a corresponding symbol table for that assembler segment. For detailed information about segments, modules, and global parts, see section 3, page 33.

`-G`	Forces all symbols to *symfile* if the `-g` option is used. Usually, nonreferenced symbols are not included.
`-C` *cpu*	Generates code for the CPU specified. By default, code is generated for the machine specified by the `TARGET` environment variable. If the `TARGET` environment is not set, code is generated for the characteristics of the host machine. *cpu* has one of the following syntaxes:

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

primary — *primary* can be one of the following Cray Research systems:

cray-c90	CRAY C90 series
cray-j90	CRAY J90 series
cray-ts	CRAY T90 series
cray-ymp	CRAY Y-MP series

charac — Specifies the features of the *primary* computer.

Cray PVP systems permit you to specify the logical and numeric traits shown in Table 1.

Table 1. Logical and numeric traits

Traits	Description
Logical	
`avl`	Additional vector logical
`noavl`	No additional vector logical
`bdm`	Bidirectional memory
`nobdm`	No bidirectional memory
`bmm`	Bit matrix multiply (BMM), only on machines supporting BMM hardware.
`nobmm`	No bit matrix multiply
`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
`ieee`†	CRAY T90 system with IEEE floating-point hardware
`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

† The `ieee` characteristic can be used to specify that code be generated to run on a CRAY T90 system with IEEE floating-point hardware; however, generating code that runs on a Cray PVP system that uses Cray floating-point arithmetic from a CRAY T90 system with IEEE floating-point hardware is not supported.

Table 1. Logical and numeric traits
(continued)

Traits	Description
`vrecur`	Vector recursion
`novrecur`	No vector recursion
Numeric	
`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 pseudo instructions regardless of the location field name.

`-H` Disables all list pseudo instructions regardless of the location field name.

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

- List of names separated by a comma
- List of names enclosed in double quotation marks and separated by a comma and/or one or more spaces

`-I` *options* Specifies a list of options. You can specify a list of more than one option without intervening blanks. You cannot specify conflicting options (for example, the same character in uppercase and lowercase) in the same `-I` list. For valid options, see Table 2.

†† *n* represents an unsigned decimal number

Table 2. List options

Options	Description
`b`	Enables macro, opdef, dup, and echo expansion (binary only)
`B`†	Disables macro, opdef, dup, and echo expansion (binary only)
`c`	Enables macro, opdef, dup, and echo expansion (conditionals)
`C`†	Disables macro, opdef, dup, and echo expansion (conditionals)
`d`	Enables dup and echo expansion
`D`†	Disables dup and echo expansion
`e`†	Enables edited statement listing
`E`	Disables edited statement listing
`l`	Enables listing control pseudo instructions
`L`†	Disables listing control pseudo instructions
`m`	Enables macro and opdef expansions (binary only)
`M`†	Disables macro and opdef expansions (binary only)
`n`†	Enables nonreferenced local symbols included in the cross-reference
`N`	Disables nonreferenced local symbols included in the cross-reference
`p`	Enables macro, opdef, dup, and echo expansion of pre-edited lines
`P`†	Disables macro, opdef, dup, and echo expansion of pre-edited lines

† Denotes default option

Table 2. List options
(continued)

Options	Description
`s`†	Enables source statement listing
`S`	Disables source statement listing
`t`	Enables text source statement listing
`T`†	Disables text source statement listing
`x`†	Enables cross-reference listing
`X`	Disables cross-reference listing

`-m` *mlevel* Specifies the level of the output listing, the message listing, and the standard error file. *mlevel* can be `comment`, `note`, `caution`, `warning`, or `error`.

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

`-M` Enables flagging of possible CRAY C90 series and CRAY J90 series bidirectional memory conflicts. Requires `-m` to be set to `comment`, `note`, or `caution`.

`-n` *number* Maximum number of messages that will be inserted into the output listing, the message listing, and the standard error file. *number* must be 0 or greater; the default is 100.

`-f` Enables the new statement format. By default, the old format is used when targeting for a CRAY Y-MP system; otherwise, the new format is used. Statement format reverts to the format that is specified on the invocation statement at the end of each assembler segment.

† Denotes default option

`-F`	Disables the new statement format. By default, the old format is used when targeting for a CRAY Y-MP system; otherwise, the new format is used. Statement format reverts to the format specified on the invocation statement at the end of each assembler segment.
`-j`	Enables editing; the default is enabled. Editing status reverts to the status specified on the invocation statement at the end of each assembler segment.
`-J`	Disables editing; the default is enabled. Editing status reverts to the status specified on the invocation statement at the end of each assembler segment.
`-U`	Forces the conversion of source code to uppercase. Quoted strings are embedded micros and are protected. Both new and old format statement types are supported.
`-V`	Causes the version number of the assembler being run and other statistical information (comment-level diagnostic messages) to be written to the standard error file.
file	File that will be assembled; all options must precede the file name argument.

Interactive assembly
2.2

To assemble and execute a CAL program interactively, enter the following commands:

as *myfile*.s

segldr *myfile*.o

a.out

The commands are described as follows:

Command	Description
`as`	Assembles file *myfile*`.s` and creates file *myfile*`.o`
`segldr`	Links and loads the assembled program found in *myfile*`.o` and creates the executable file `a.out`
`a.out`	Executes the executable file `a.out`

For a description of these and other commands, see the *UNICOS User Commands Reference Manual*, publication SR–2011.

The UNICOS environment
2.3

The following subsections describe aspects of the UNICOS environment. How the environment is set depends on the type of shell being used.

`LPP` *environment variable*
2.3.1

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

To set the `LPP` environment variable and assemble multiple source files when using the C shell, enter the following commands:

```
setenv LPP n
as filenamea.s
as filenameb.s
.
.
.
```

To set the LPP environment variable and assemble a source file with a single command line when using the standard shell, enter the following command:

LPP=*n* as *filenamex*.s

If you specify the LPP shell variable on the same line as the as(1) command line, the number of lines per page assigned by the LPP shell variable is restricted to that particular as instruction.

To set the LPP environment variable and assemble multiple source files when using the standard shell, enter the following commands:

LPP=*n*

export LPP

as *filenamea*.s

as *filenameb*.s

.

.

.

If you specify the LPP shell variable as a separate entry and then export it, all assemblies that follow use the page length specified by that LPP shell variable for output and message listings.

In the preceding examples, *n* is a decimal number in a valid range of 4 through 999 (the default is 55) that represents the page length used in output listings and *filenamea, filenameb...* represent the names of the source files being assembled.

Note: If *n* is outside of the valid range, the page length is set to the default.

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

```
LPP=45

export LPP

as srca.s

as srcb.s
```

In the following example, the page length for `srcd.s` is 45. However, the page length for `srce.s` reverts to 64 because the second `LPP` shell variable is restricted to the assembly of `srcd.s`:

```
LPP=64

export LPP

LPP=45 as srcd.s

as srce.s
```

TMPDIR ***shell variable*** 2.3.2

The `TMPDIR` shell variable specifies a directory used by the assembler for its temporary file. If the directory is not specified or is specified incorrectly, the assembler uses the system default. The default is site-specific.

To set the `TMPDIR` environment variable and assemble a source file when using the C shell, enter the following commands:

```
setenv TMPDIR dir_name

as filenamea.s
```

To set the `TMPDIR` environment variable and assemble a source file with a single command line when using the standard shell, enter the following command:

```
TMPDIR=p as filenamex.s
```

If you specify the `TMPDIR` shell variable on the same line as the `as`(1) command line, the temporary directory assigned by the `TMPDIR` shell variable affects only that particular `as` instruction.

To set the `TMPDIR` environment variable and assemble multiple source files when using the standard shell, enter the following commands:

```
TMPDIR=p

export TMPDIR

as filenamea.s

as filenameb.s

.

.

.
```

If you specify the `TMPDIR` environment variable as a separate entry and then export it, all assemblies that follow use the temporary directory specified by that `TMPDIR` shell variable for temporary files.

In the preceding example, *p* specifies the directory path used for the assembler's temporary file and the *filenamea, filenameb...* variables represent the names of the UNICOS files that are being assembled.

In the following example, `/tmp` is the directory that CAL uses for its temporary file:

```
TMPDIR=/tmp

export TMPDIR

as srca.s

as srcb.s
```

In the following example, the temporary directory used for `srcd.s` is the current working directory (`.`). The temporary directory for `srce.s`, however, reverts to `/usr/tmp`, which is used because the second `TMPDIR` environment variable is associated only with the assembly of `srcd.s`:

```
TMPDIR=/usr/tmp

export TMPDIR

TMPDIR=.  as srcd.s

as srce.s
```

`MSG_FORMAT` *error message format* 2.3.3

The `MSG_FORMAT` environment variable controls the format of error messages received from programs that use the `catmsgfmt`(3) message formatting routine. For more information, see the `explain`(1) man page.

`TARGET` shell variable 2.3.4

The `TARGET` environment variable determines the characteristics of the machine the code is generated for. To initialize the `TARGET` environment variable in the C shell, enter the following:

setenv TARGET *cpuname*

To initialize the `TARGET` environment variable in the standard shell, enter the following:

```
export TARGET
```

The format to set up or change the `TARGET` environment variable in the standard shell is as follows:

TARGET=[*cpuname*]{,[*charac*]}

If the `TARGET` environment variable is not set, code is generated using the characteristics of the host machine. The options for *cpuname* and *charac* may be found in subsection 2.1, page 12. For more information, see the `target`(1) man page.

> **Note:** Targeting a CRAY T90 with IEEE floating-point hardware from any other Cray PVP system is supported, however; targeting a Cray PVP system that uses Cray floating-point arithmetic from a CRAY T90 with IEEE floating-point hardware is not supported.

Binary definition files
2.4

CAL allows the 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*. Binary definition files are analogous to libraries and are one of the following types:

- System-defined
- User-defined

The system-defined binary definition file is `/lib/asdef`. CAL accesses the system-defined binary definition file automatically unless the assembler is directed otherwise. Binary definition files contain commonly used symbols, macros, opdefs, opsyns, and micros. For information about available macros and opdefs, see the *UNICOS Macros and Opdefs Reference Manual*, publication SR–2403.

> **Note:** System- and user-defined binary definition files are identical in all respects. Both types of files are created and used in exactly the same manner. In this manual, they are treated as separate entities to encourage you to define binary definition files that meet your particular programming requirements.

You can create user-defined binary definition files by using either of the following methods:

- Copying the system-defined binary definition files and then modifying the new file either by adding new definitions or by redefining existing definitions.
- Disabling the recognition of system-defined binary definition files and accumulating the defined sequences entirely from an assembler source program. For more information, see subsection 2.1, page 12.

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 specified by the `-b` option.

Lines or sequences of code assembled and stored in a binary definition file, can be accessed without reassembly. This means accessing a binary definition file directly saves assembler time.

The following subsections describe defining, creating, and using binary definition files.

Defining a binary definition file 2.4.1

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 segments and from any currently accessed binary definition files. Typically, binary definition files are created from source programs that include one segment that contains a global part, but has no program module (see Figure 4, page 28).

Additions can be made to binary definition files from assembler source programs that include program modules, however, not all lines or sequences of code in the global part are added.

> **Note:** Under no circumstance is any line or sequence of code added to a binary definition file from an assembler program module. All additions to binary definition files come from the global part of the segment.

Binary definition files are composed of lines or sequences of code classified as follows:

- Symbols
- Macros
- Opdefs
- Opsyns
- Micros

Each line or sequence of code added to a binary definition file must be in one of these classes and must satisfy the requirements for that particular class.

Symbols 2.4.1.1

CAL accumulates the 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 cannot be redefinable.

  To be included in a binary definition file, a symbol must be defined with the = (equates) pseudo instruction. Symbols defined with the SET or MICSIZE pseudo instruction are redefinable; therefore, they are not included in a binary definition file.

- Symbols 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 preceding requirements and includes them in the creation of a binary definition file. In Figure 4, page 28, SYM1, SYM3, and SYM4 meet the requirements and are included. SYM2 (defined in the module), SYM5 (redefinable), and %%SYM6 (begins with %%) do not meet the requirements and are not included.

Macros 2.4.1.2

CAL accumulates the 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.

Opdefs 2.4.1.3

CAL accumulates opdefs (operation definitions) 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.

Opsyns 2.4.1.4

CAL accumulates opsyns (operation synonyms) 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.

Micros 2.4.1.5

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. A micro must be defined using the `CMICRO` pseudo instruction to be included in a binary definition file.

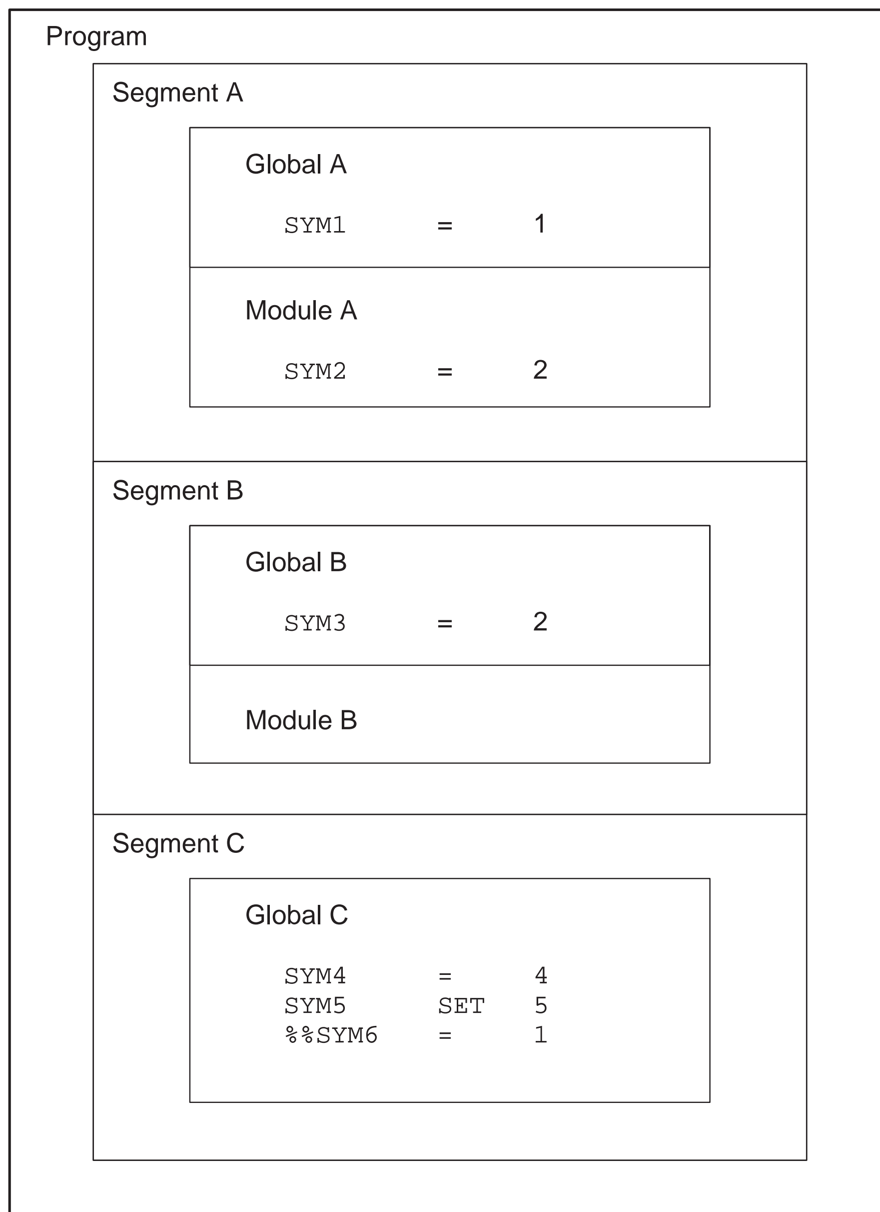


Figure 4. CAL program structure

Creating binary definition files
2.4.2

You can create your own binary definition files containing information related to both the UNICOS operating system and Cray Research hardware. Keep in mind that system dependencies included in these binary definition files may not be portable between UNICOS versions or different hardware platforms.

To create binary definition files under UNICOS, include the `-b` and `-c` options on the `as`(1) command line. The `-b` option accepts a list of files separated by commas or a list of files enclosed in double quotation marks and separated by spaces or commas as arguments.

In the following example, the default system-defined binary definition file `/lib/asdef` and user-defined binary definition files `ourdeffile` and `mydeffile` are 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 called `mynewfile` is defined and created by including the `-c` option.

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

In CAL, the default binary definition file (`/lib/asdef`) is available unless suppressed by including the `-B` option. If not suppressed, `/lib/asdef` is the first binary definition file read. Any other binary definition files specified following the `-b` option are processed in the order specified. The following command line suppresses `/lib/asdef` and makes `mynewfile` the only available binary definition file:

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

The following command line suppresses `/lib/asdef` and takes only the accumulated symbols, macros, opdefs, opsyns, and micros from the global parts of the program segments from the current source program being assembled and enters them into the binary definition file `mynewfile`:

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

To use the system-defined binary definition file and specify the user-defined binary definition file created using the preceding options, use the following command line in subsequent assemblies:

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

Using binary definitions files 2.4.3

Binary definition files provide access to previously assembled lines or sequences of code. To access binary definition files, include the `-b` option on the `as`(1) command line. When binary definition files are accessed, they are checked for the following:

- CPU compatibility
- Multiple references to the same definition

CPU compatibility checking 2.4.3.1

CAL permits access to any previously defined file with one restriction. Binary definition files are marked with the CPU type for which they were created. Binary definition files created on one Cray PVP system is not necessarily compatible with all Cray PVP systems. If a binary definition file is not compatible with the system you are using, 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 trying to use it. Some CAL pseudo instructions have restricted use based on hardware and software requirements. The binary definition file compatibility check prevents the mixing of binary definition files and ensures that hardware and software restrictions are not violated.

Multiple references to a definition 2.4.3.2

CAL checks for multiple references to definition names for macros and opsyns, location field names for symbols and micros, and syntax for opdefs. The following subsections describe how multiple references to a definition are resolved.

Symbols 2.4.3.2.1

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 2.4.3.2.2

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 functional name of the macro 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 2.4.3.2.3

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 and CAL 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 message shown above.

Opsyn 2.4.3.2.4

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, CAL disregards the duplicate definition and makes one entry for the opsyn from the binary definition files. If the functional name for an opsyn is used more than once and the definitions are not identical, 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 message as shown above. Pseudo 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
2.4.3.2.5

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, 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
```

The CAL Program [3]

This section describes the organization of a CAL program and how each component functions within the program. A CAL program can contain any or all of the following components:

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

The following subsections describe each of these components.

Program segment
3.1

A CAL program consists of zero or more segments. A CAL program with zero segments consists of one or more empty files. A file that contains 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 5, page 34, illustrates the organization of a CAL program.

Program module
3.1.1

A *program module* is the main body of code and resides between the `IDENT` and `END` pseudo instructions. (For more information on pseudo instructions, see subsections 3.4.1.2, page 47, and section 5, page 117.) The `IDENT` pseudo instruction marks the beginning of a program module. The `END` pseudo instruction marks the end of a module and always terminates a segment. Any definitions between these two pseudo instructions apply only to the program module in which the definition resides.

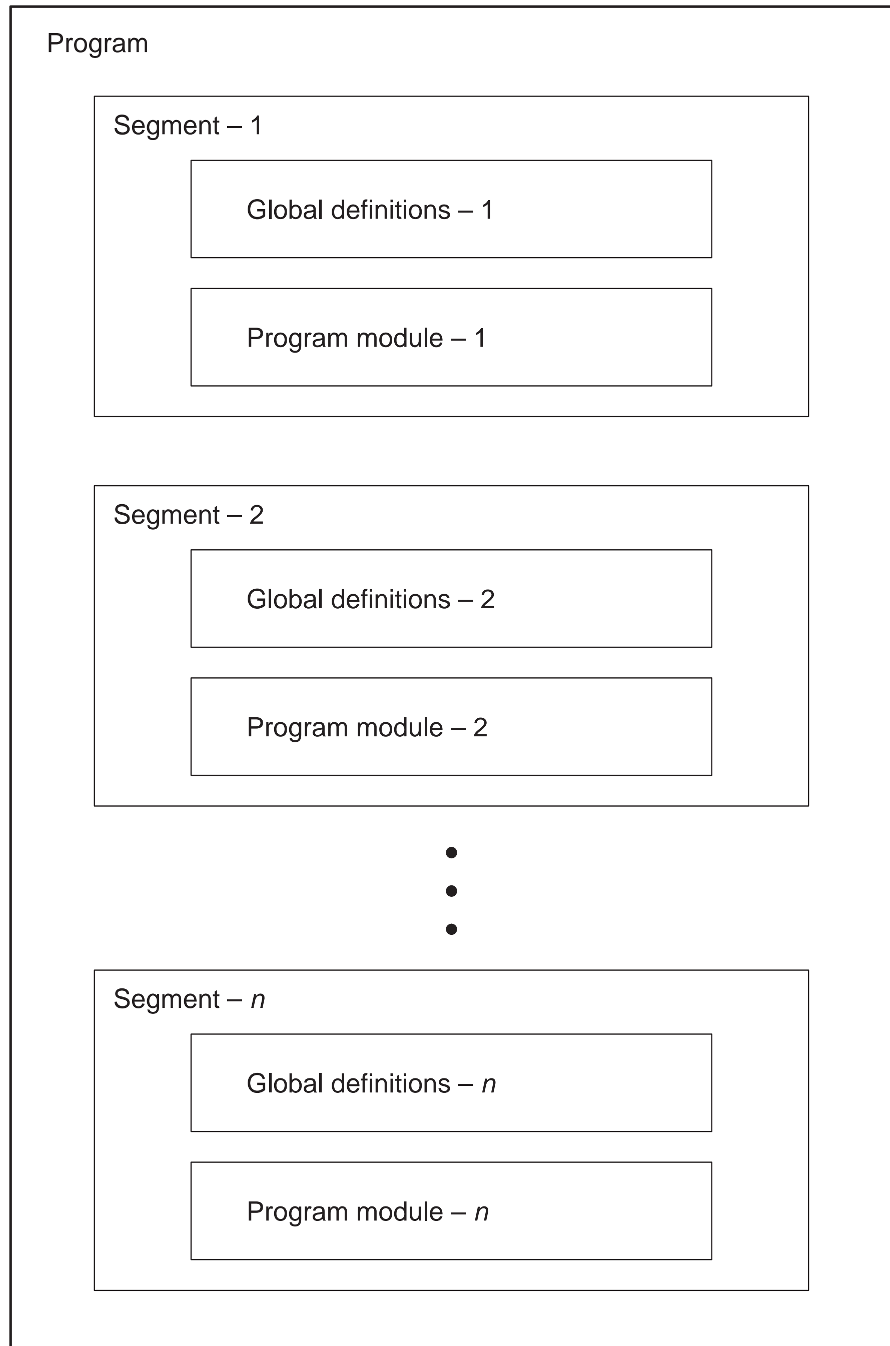


Figure 5. CAL program organization

Global definitions
3.1.2

Definitions that occur before the first IDENT pseudo instruction or between the END pseudo instruction that terminates one program module and the IDENT that begins the next program module are *global definitions*. They can be referenced without redefinition from within any of the program segments that follow the definition.

CAL recognizes global definitions to be sequences of instructions that do not generate code. They define and assign values to symbols, macros, opdef instructions, and micros. (For more information on opdefs, macros, and micros, see section 5, page 117.)

Redefinable micros, redefinable symbols, and symbols of the form %%*x*; where *x* is 0 or more identifier-characters are exceptions. Although 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 encountered, and they are not included in the cross-reference listing. Symbols defined within the global definitions area cannot be qualified (see subsection 4.3.1, page 70).

The following example illustrates global definitions:

```
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 TEST 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 TEST 2

       IDENT  TEST3        ; Beginning of segment 3 and module TEST 3
       S1   SYM1           ; Register S1 gets 1
       S2   SYM2           ; Error:  not defined
       S3   %%SYM3         ; Error:  not defined
       END                 ; End of segment 3 and module TEST 3
```

Source statement
3.2

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 executable program.)

Formal parameters, symbols, names, pseudo instructions, and macro names are case-sensitive. To be recognized, subsequent references to a previously defined formal parameter, symbol, name, or functional unit 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

The following rules govern the use of uppercase and lowercase characters in CAL statements:

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

Although CAL source statements are essentially free field, formatting conventions provide more uniform and readable listings. CAL supports two formatting conventions, the new format and the old format. A blank character is used to separate fields in the old format. In the new format, you can use either a blank or a tab to separate fields.

New format 3.2.1

The new format is specified by either the `FORMAT` pseudo instruction or the `–f` parameter of the CAL invocation statement. For more information on the `–f` parameter, see subsection 2.1, page 12.

A source statement that uses the new format consists of the following fields:

- Location
- Result
- Operand
- Comment

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

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

Location field 3.2.1.1

The content of the location field depends 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 value of the location counter.

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

Result field 3.2.1.2

The content of the result field depends on the particular instruction. The result field of pseudo instructions and macro instructions must match existing functionals. Machine or opdef instructions can contain one, two, or three subfields.

The subfield can be empty, contain expressions, or consist of register designators or operators. (Expressions, register designators, and operators are described in section 4, page 61.) The result field begins with the first nonblank or nontab character following a location field that is not empty and usually

ends with one or more blanks, one or more tabs, or a semicolon. If column l is empty, the result field can begin in column 2 or subsequent columns. A blank result field following a location field produces a listing message.

Operand field
3.2.1.3

Before the operand field can be specified, it must be preceded by a result field. For functionals (pseudo instructions and macro names), the operand field depends on the functional specified in the result field.

If the instruction is a symbolic machine instruction, the operand field contains the operation being performed. However, it can 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 empty, contain zero or more expressions, or consist of register designators and operators.

Usually, the operand field begins with the first nonblank or nontab character following a result field that is not empty and ends with one or more blank characters, one or more tab characters, or a semicolon.

Comment field
3.2.1.4

The comment field contains an explanation of the source statement and does not generate code. The comment field is optional and can be specified with an asterisk or a semicolon. A semicolon comment can be in any column, including column 1. If an asterisk is used to indicate a comment, it must appear in column 1. Generally, a comment that begins in column 1 is specified by using an asterisk and a comment that begins in any other column is specified by using a semicolon. If a semicolon is specified with nothing preceding it, the line is treated as a null instruction followed by a comment. Usually, comment fields are not edited. For more information about editing comment fields, see subsection 3.3, page 41.

The following example illustrates the use of the comment field:

```
       ident  test1
*Asterisk in column 1 denotes comment line
                               ; Semicolon begins comment
       end    test1
```

Old format
3.2.2

The old format is specified by either the `FORMAT` pseudo instruction or the `-F` parameter of the CAL invocation statement. For more information on the `-F` parameter, see subsection 2.1, page 12.

A source statement that uses the old format consists of the following fields:

- Location
- Result
- Operand
- Comment

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

Location field
3.2.2.1

The content of the location field depends 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 value of the location counter.

If the location field contains an asterisk (in column 1 only), that line is identified as a comment line. The location field is not used by all instructions. It begins in column 1 or 2 (old format) and is terminated by a blank character.

A comma can be used for a continuation line. For more information, see subsection 3.3, page 41.

Result field
3.2.2.2

The result field begins with the first nonblank character following the location field and ends with one or more blanks or the end of the statement. If the location field terminates before column 33, the result field must begin before column 35; otherwise, the field is considered empty. If the location field extends beyond column 32, however, the result field must begin after not more than one blank separator and can begin after column 35.

Operand field
3.2.2.3

The operand field begins with the first nonblank character following a result field that is not empty and ends with one or more blanks or the end of the statement. 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 begin after not more than one blank separator and can begin after column 35.

Comment field
3.2.2.4

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 subsection 3.3, page 41.

The following example illustrates the use of the comment field:

```
         IDENT
* An asterisk comment must begin in column 1.
```

Statement editing
3.3

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 assembler performs the following types of statement editing:

- 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 edited a second time.

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 using the `EDIT` pseudo instruction or by including the `-J` parameter in the invocation line of the assembler.)

The edit invocation statement option does not affect appending, continuation, and the processing of comments.

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

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

Character	Edit	Description
`*`	No	Comment line; (asterisk) unaffected by the `EDIT` pseudo instruction on the invocation statement option (new or old format).
`;`	No	Comment line; (semicolon) unaffected by the `EDIT` pseudo instruction on the invocation statement option (new or old format).

Note: When CAL edits `"$CMNT"`, `"$MIC"`, `"$CNC"`, or `"$APP"`, the string name and the pair of double quotation marks (`" "`) is replaced by a previously defined string. For example, when CAL edits `"$CMNT"`, a semicolon is substituted for the micro name `$CMNT` and the double quotation marks (`" "`). After the substitution occurs, the semicolon is not edited again 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.

Micro substitution
3.3.1

You can assign a micro name to a character string. You can refer to that character string in subsequent statements 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 with the character string that comprises the micro. For more information on micros, see subsection 3.5, page 48.

Concatenate
3.3.2

The concatenate feature combines characters connected by the underscore (_) character. CAL examines each line for the underscore character and deletes each occurrence of the underscore. 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.

Append
3.3.3

The append feature combines source statements that continue for more than one line. It is available only when the new format is specified. The exact number of lines that CAL can append depends on memory limitations.

The append symbol is a circumflex (^) and appends one line to another. It can be used in any column on any line. If more than one circumflex exists, the first instance is used.

When the current line contains a circumflex, CAL appends the first nonblank or nontab character and all characters that follow from the next line to the current line. The characters are appended at the position in the current line that contains the circumflex; the circumflex and any characters that follow it on the current line are replaced.

Continuation
3.3.4

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

Comment
3.3.5

A semicolon (;) in any column (new format only) or an asterisk (*) in column 1 indicates 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 used. In the new format, comment statements with semicolons or asterisks are not appended.

> **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. Because 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.

The following example illustrates the use of comment statements in a macro:

```
        MACRO
        EXAMPLE
* This comment is not included in the definition.
_* This comment is included in the definition.
SYM     =    1
EXAMPLE ENDM
```

The macro in the preceding example is expanded as follows:

```
        LIST          LIS,MAC
        EXAMPLE            ;Macro call
* This comment is included in the definition.
SYM     =    1
```

Actual statements and edited statements
3.3.6

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 of 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.

This following example shows an actual statement:

```
LOC      MCALL          ARG1,^
                        ARG2,^
                        ARG3,^
                        ARG4,^
                        ARG5,^
```

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 the preceding example:

```
LOC      MCALL          ARG1,ARG2,ARG3,ARG4,ARG5
```

In the following example, the actual statement has no corresponding edited statement:

```
         ENTER          ARG1,ARG2,ARG3         ; Comments
```

Instructions
3.4

CAL recognizes two types of instructions:

- Assembler-defined
- User-defined

Assembler-defined instructions include machine and pseudo instructions. *User-defined instructions* are defined by the user.

Assembler-defined instructions
3.4.1

Two types of assembler-defined instructions are available in CAL:

- Machine instructions
- Pseudo instructions

Machine instructions 3.4.1.1

Machine instructions manipulate data by performing functions such 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 each instruction can contain an optional symbol. If an optional symbol is included, it is not redefinable, has a value equal to the value of the current location counter and an address attribute of parcel, and its relative attribute is equal to the relative attribute of the current location counter (that is, absolute, immobile, or relocatable). For more information about symbols and expression evaluation, see section 4, page 61.

Machine instruction syntax is uniquely defined by the contents of the result field alone or the result and operand fields together. The optional location field represents the logical memory location of the instruction.

Each Cray Research system has its own set of machine instructions. Appendix D, page 359, and appendix E, page 369, contain tables of these machine instructions. For more detailed descriptions of the instruction sets for each system, see the system programmers reference manual for the particular system.

Pseudo instructions 3.4.1.2

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.

Section 5, page 117, describes the use of pseudo instructions and appendix A, page 189, describes individual pseudo instructions and their formats.

User-defined instructions 3.4.2

The CAL assembler lets you identify a sequence of instructions that will 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 are classified as either permanent or 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.

Micros
3.5

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, page 124) assign the name to the character string.

Refer to a micro by enclosing the micro name in double quotation marks (“ ”) anywhere in a source statement other than within a comment. If column 72 of a line is exceeded because of a micro substitution, the assembler creates additional continuation lines. No replacement occurs if the micro name is unknown or if one of the quotation marks is omitted.

When a micro is edited, the source statement that contains the micro is changed. Each substitution produces one of the following cases:

- The length of the micro name and the pair of quotation 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 quotation 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 quotation marks and the predefined substitute string.

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

In the following example, the length of the micro name (including quotation marks) 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:

```
"PFX"TAG   S0              S1    ; The location of S0 and S1 on the
                                 ; source statement is unchanged
```

When the line is interpreted, CAL substitutes `EQUAL` for `"PFX"`, producing the following line:

```
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 (including quotation marks) 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:

```
"PFX"TAG   S0              S1    ; Because 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 `LESS` for `"PFX"`, producing the following line:

```
LESSTAG  S0              S1    ; Because 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 (including quotation marks) 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:

```
"pfx" tag  S0            S1    ; Because greater is two characters
                               ; longer than 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"`. Because the predefined substitute string is 2 characters longer than micro name, the fields to the right of the substitution are shifted 2 characters to the right, producing the following statement:

```
greatertag   S0          S1    ; Because greater is two characters
                               ; longer than the micro string name
                               ; "pfx", the values in the result and
                               ; operand fields are shifted
```

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 *embedded micros*. An embedded micro consists of a micro name included between a left ({) and a right brace (}) and is specified as follows:

{*microname*}

When a micro that contains one or more embedded micros is encountered, CAL edits all 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 8 characters). When an illegal micro is encountered, CAL issues an appropriate message and terminates the editing of the micro. An embedded micro also can contain one or more embedded micros.

The following example includes valid and not valid defined embedded micros

```
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" - an explicit reference.
* Some string - an explicit reference†
* "array" "index" - not valid, because "array" was not defined.
* "array"1 - not valid, 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.†
```

† Edited by CAL

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 capability of CAL on embedded micros:

```
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 a warning- or error-level listing message when an invalid micro name is specified. If a micro name is recognized as invalid before editing begins, a warning-level message is issued. If an embedded micro has been edited and the resulting string is not a valid micro name, an error-level listing message is issued.

† Edited by CAL

The following examples demonstrate how CAL assigns levels to messages when a micro that is not valid is encountered

```
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 warning-level
* listing message is issued:
*    "identity9{null}" - This is a not valid micro.
*    "identity9 - This is a not valid micro.†

* The following micro is not valid, because the maximum micro name
* length of eight characters is exceeded.  When a micro name is
* identified as being not valid after editing occurs, an error-level
* listing message is issued:
*    "id{null}entity9{null}" - This is a not valid micro.
*    "identity9" - This is a not valid micro.†
```

Sections
3.6

A CAL program module can be divided into blocks of memory called *sections*. By dividing a module into sections, you can conveniently separate sequences of code from data. As the assembly of a program progresses, 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.

Use the main and literals sections for implicitly assigned code. CAL maintains a stack of section names assigned by the `SECTION` pseudo instruction. All sections except stack sections are passed directly to the loader.

† Edited by CAL

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 assign code explicitly to a section, use the `SECTION` pseudo instruction. You can specify the `SECTION` pseudo instruction for any Cray PVP system.

Local sections 3.6.1

A local section is a block of code that is usable 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 instruction

When a `SECTION` pseudo instruction is used, every `SECTION` type except `COMMON`, `DYNAMIC`, `TASKCOM`, and `ZEROCOM` is local. For more information about `SECTION` types, see the `SECTION` pseudo instruction in subsection 5.4, page 120.

Main sections 3.6.1.1

The main section is initiated by the `IDENT` pseudo instruction 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.

Generally, sections may not have names but must be assigned types and locations. The default name of the main section is always empty. The defaults for type and location are `MIXED` and `CM`, respectively. For more information about the `MIXED` and `CM` section names, see the `SECTION` pseudo instruction in subsection 5.4, page 120.

Literals section 3.6.1.2

The first use of a literal value in an expression causes the assembler to store the data item in a literals section. 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. The assembler supports the literals section as a constant section. For more information about literals, see subsection 4.4.3, page 85.

Sections defined by the SECTION *pseudo instruction* 3.6.1.3

When a SECTION pseudo instruction is used, all code generated or memory reserved (other than literals) between occurrences of SECTION pseudo instructions is assigned to the designated section.

Until the first SECTION pseudo instruction is specified, the main section is used. If you specify the ORG pseudo instruction, an exception to these conditions can occur. Specifying the ORG pseudo instruction may cause the placement of code or memory reservations to be different from the currently specified working section.

The SECTION pseudo instruction is recommended for use with all Cray PVP systems because it has all of the same capabilities as the BLOCK and COMMON pseudo instructions.

When a section is released, 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. For more information, see the SECTION pseudo instruction in appendix A, page 189.

Common sections 3.6.2

When a SECTION pseudo instruction is used with a type of COMMON, DYNAMIC, ZEROCOM, or TASKCOM, all code generated (other than literals) or memory reserved between occurrences of SECTION pseudo instructions is assigned to the designated common, dynamic, zero common, or task common section. The SECTION pseudo instruction replaces the COMMON pseudo instruction. You can use SECTION in any of the ways that COMMON was used previously.

At program end, each common section is identified to the loader by its SECTION name and is available for reference by another program module. If you specify the ORG pseudo instruction, an exception to these conditions can occur. 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 specified, the identifier in the location field that names the section must be unique within the module in which it is defined. When a section is assigned a type (`COMMON`, `DYNAMIC`, `ZEROCOM`, or `TASKCOM`) that differs from the type of a previously defined section, it cannot be assigned the name of a previously defined section within the same module.

Section stack buffer 3.6.3

CAL maintains a stack buffer that contains a list of the sections 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. You also can use the `BLOCK` and `COMMON` pseudo instructions to name sections.

CAL remembers the order in which sections are specified. An entry is deleted from the list each time a `SECTION` pseudo instruction contains an asterisk `(*)`. When an entry is deleted, the name, location, and type of the previously specified section is enabled.

The first section on the list is the last section that will be deleted from the list. If the program contains more `SECTION *` instructions than there are entries, the assembler uses the main section. (The `BLOCK *` and `COMMON *` instructions replace the current section with the most recent previous section that was specified by the `BLOCK` and `COMMON` pseudo instructions.)

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 following example illustrates section specification and deletion and indicates the current section. The example includes the `QUAL` pseudo instruction. For a description of the `QUAL` pseudo instruction, see appendix A, page 189.

```
      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             ; SYM2 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 QNAMEl 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
                             ; QNAMEl even though QNAME1 is not the
                             ; current qualifier in effect.
      QUAL     *             ; QNAME2 is removed as the current
                             ; qualifier name.
SYM7  =        6             ; SYM7 is the fourth entry in the symbol
                             ; table for QNAME1.
SYM8  =        7             ; Second entry in the symbol table for
                             ; unqualified symbols.
```

Origin counter 3.6.3.1

The *origin counter* controls the relative location of the next word that will be assembled or reserved in the section. You can reserve blank memory areas 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. To obtain the word-address value of the origin counter, use `W.*O`. For more information about the special elements and `W.` prefix, see subsection 4.5, page 89.

Location counter 3.6.3.2

Usually, the *location counter* is the same value as the origin counter and the assembler uses it to define symbolic addresses within a section. The counter is incremented when the origin counter is incremented. 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 with the current parcel-address value of the location counter for the section in use. To obtain the word-address value of the location counter, Use `W.*` (see subsection 4.5, page 89).

Word-bit-position counter 3.6.3.3

As instructions and data are assembled and placed into a word, CAL maintains a pointer that indicates the next available bit within the word currently being assembled. This pointer is known as the *word-bit-position counter*. It is 0 at the beginning of a new word and is incremented by 1 for each completed bit in the word. Its maximum value is 63 for the rightmost 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. You can alter this normal advancement by using the `BITW`, `BITP`, `DATA`, and `VWD` pseudo instructions.

Force word boundary 3.6.3.4

If either of the following conditions are true, the assembler completes a partial word and sets the word-bit-position and parcel-bit-position counters to 0:

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

Parcel-bit-position counter 3.6.3.5

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 the *parcel-bit-position counter*. It is 0 at the beginning of a new parcel and advances by 1 for each completed bit in the parcel. The maximum value is 15 for the rightmost 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 is set to 0 following assembly of most instructions. The pseudo instructions `BITW`, `BITP`, `DATA`, and `VWD` can cause the counter to be nonzero.

Force parcel boundary 3.6.3.6

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

Cray Assembly Language [4]

This section presents the general rules and statement syntax for Cray Assembly Language (CAL). 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

Register designators
4.1

Register designators are used in symbolic machine instructions and opdefs to specify the register to be used for an operation. CAL accepts register mnemonics specified in uppercase, lowercase, or mixed case. Each Cray PVP system supports all or a subset of simple and complex registers.

Complex registers are members of a set of registers that are identical in function and architecture. The set of registers is identified by a letter. The specific register within the set is specified by an octal number up to 4 octal digits in length or a constant. For example, you specify register `S1` from the set of S registers.

The following example illustrates complex register designation:

```
      A1     SyM          ; CAL permits mixed case 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.
```

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

The following example illustrates simple register designation:

```
      S1     RT           ; Register S1 gets the contents of the RT
                          ; register
```

Table 3 lists register designations for CRAY Y-MP, CRAY C90, CRAY J90, and CRAY T90 series systems. By convention, B and T registers are written using two octal digits.

Note: A, B, and SB registers are expanded to 64 bits (32 bits in C90 mode) on CRAY T90 systems. Several new instructions use the 64-bit A registers for logical and shift operations. See appendix E, page 369, for more information on the CRAY T90 instruction set.

Table 3. Register designations

Register type	CRAY Y-MP Number Mnemonic Size (in bits)	CRAY C90 Number Mnemonic Size (in bits)	CRAY J90 Number Mnemonic Size (in bits)	CRAY T90 Number Mnemonic Size (in bits)
Data registers (A registers)	8 `A0 – A7` 32	8 `A0 – A7` 32	8 `A0 – A7` 32	8 `A0 – A7` 64
Data registers (B registers)	64 `B00 – B77` 32	64 `B00 – B77` 32	64 `B00 – B77` 32	64 `B00 – B77` 64

Table 3. Register designations
(continued)

Register type	CRAY Y-MP Number Mnemonic Size (in bits)	CRAY C90 Number Mnemonic Size (in bits)	CRAY J90 Number Mnemonic Size (in bits)	CRAY T90 Number Mnemonic Size (in bits)
Scalar registers (S registers)	8 `S0 – S7` 64	8 `S0 – S7` 64	8 `S0 – S7` 64	8 `S0 – S7` 64
Transfer registers (T registers)	64 `T00 – T77` 64	64 `T00 – T77` 64	64 `T00 – T77` 64	64 `T00 – T77` 64
Vector registers (V registers)	8 `V0 – V7` 64 (64 elements)	8 `V0 – V31` 64 (128 elements)	8 `V0 – V31` 64 (64 elements)	8 `V0 – V63` 64 (128 elements)
Shared address registers (SB registers)	8/cluster `SB0 – SB7` 32	8/cluster `SB0 – SB7` 32	8/cluster `SB – SB7` 32	16/cluster `SB0 – SB7` 64
Semaphore register (SM register)	32/cluster `SM0 – SM37` 1	32/cluster `SM0 – SM37` 1	32/cluster `SM0 – SM37` 1	64/cluster `SM0 – SM77` 1
Status registers (SR registers)	1 `SR` 32	8 `SR0 – SR7` 64	1 `SR` 32	8 `SR0 – SR7` 64
Shared scalar registers (ST registers)	8 `ST0 – ST7` 64	8 `ST0 – ST7` 64	8 `ST0 – ST7` 64	16 `ST0 – ST15` 64
Channel address register (CA register)	1/channel `CA` 32	1/channel `CA` 32	1/channel `CA` 32	1/channel `CA` 64
Channel error register (CE register)	1/channel `CE` 32	1/channel `CE` 32	1/channel `CE` 32	1/channel `CE` 64

Table 3. Register designations
(continued)

Register type	CRAY Y-MP Number Mnemonic Size (in bits)	CRAY C90 Number Mnemonic Size (in bits)	CRAY J90 Number Mnemonic Size (in bits)	CRAY T90 Number Mnemonic Size (in bits)
Channel interrupt register (CI register)	1/channel `CI` 32	1 `CI` 6	1/channel `CI` 1	1/channel `CI` 32
Channel limit register (CL register)	1/channel `CL` 32	1/channel `CL` 32	1/channel `CL` 1	1/channel `CL` 32
Master clear register (MC register)	1/channel `MC` 1	1/channel `MC` 1	1/channel `MC` 1	1/channel `MC` 1
Real-time clock register (RT register)	1 `RT` 64	1 `RT` 64	1 `RT` 64	1 `RT` 64
Vector length register (VL register)	1 `VL` 7	1 `VL` 8	1 `VL` 7	1 `VL` 8
Vector mask registers (VM registers)	1 `VM` 64	2 `VM0`, `VM1` 32	1 `VM` 64	2 `VM0`, `VM1` 64
Exchange address register (XA register)	1 `XA` 8	1 `XA` 8	1 `XA` 10	1 `XA` 16
Program address register (P register)	1 `P` 24	1 `P` 32	1 `P` 24	1 `PA` 32

Vector length register (VL)
4.1.1

On CRAY C90 systems, vector registers have been increased to 128 elements and the vector mask has been increased to 128 bits. On CRAY Y-MP systems and CRAY J90 systems, the `VL` register has 7 bits. On CRAY C90 systems and CRAY T90 systems the `VL` register has 8 bits. To facilitate portable CAL code, the following symbols are available (UNICOS 7.0 and later) on all Cray PVP systems:

- `MAX$VL` (maximum vector length)
- `L2$MAX$VL` (log$_2$ of `MAX$VL` for shift and mask operations)
- `A$MAX$VL` (a micro that yields a valid A register operand equal to `MAX$VL`)

These symbols can be used in lieu of conditional code. For example:

```
C90IFC /"$CPU"/,EQ,/CRAY C90/
      S1  <7
C90ELSE
      S1  <6
C90ENDIF
      S0  S1&S2            ; Vector residual
      S2  S1&S2
      A2  S2
      $IF S0,Zero          ; If no residual
C90IFC /"$CPU"/,EQ,/CRAY C90/
      A2  D'128            ; First VL = 128
C90ELSE
      A2  D'64             ; First VL = 64
C90ENDIF
      $ENDIF
```

can be replaced with:

```
S1    <L2$MAX$VL
S0    S1&S2    ; Vector residual
S2    S1&S2
A2    S2
$IF   S0,Zero  ; If no residual
      A2  "A$MAX$VL"       ; First VL
$ENDIF
```

Vector mask register (VM)
4.1.2

The vector mask register on CRAY Y-MP systems and CRAY J90 systems is a 64-bit register, on CRAY C90 systems there are two 32-bit vector mask registers, and on CRAY T90 systems there are two 64-bit vector mask registers. The vector mask (`VM`) register is accessed through the following instructions:

```
Si    VM0   ; Get first half of VM
Si    VM1   ; Get second half of VM
VM0   Si    ; Set first half of VM
VM1   Si    ; Set second half of VM
```

If the routine does not manipulate the vector mask, no portable CAL changes are necessary. The changes necessary for a portable CAL module depend on the types of manipulations required of the vector mask. A routine that only logically combines conditions together (through AND and OR operations) can create opdefs to simulate a two-word `VM` register on all Cray PVP systems. For example:

```
C12XY  IFC   /"$CPU"/,NE,/CRAY C90/
       VM0    OPDEF
       S.REG  VM0
      *
      S.REG  VM
*
      VM0      ENDM
      VM0      OPDEF
      VM0      S.REG
*
      VM       S.REG
*
      VM0      ENDM
      VM1      OPDEF
      S.REG              VM1
*
      VM1      ENDM
      VM1      OPDEF
      VM1      S.REG
*     VM1      ENDM
C12XY  ENDIF
```

Using the above opdefs, a system that has a 64-bit `VM` register can be made to look as though it has a 128-bit `VM` register. A portable code sequence such as the following could be written:

```
S1    VM0     ; First half of VM
S2    VM1     ; Possible second half of VM
VM    V7,Z    ; Test second condition
S3    VM0     ; First half of VM
S4    VM1     ; Possible second half of VM
S1    S1!S3   ; Combine conditions
S2    S2!S4
VM0   S1      ; Set VM
VM1   S2
```

This code can then assemble on any Cray PVP system. The instructions that actually reference the second half of the `VM` register do not generate code except on CRAY C90 systems. The only extra code for other systems is the `S2 S2!S4` instruction. The `S`*i* `VM1` opdef could be modified to set `S`*i* to zero if more sophisticated vector mask operations were to be performed.

Names
4.2

Names do not have an associated value or attribute and cannot be used in expressions. Names that are 1 to 8 characters in length are used to identify the following types of information:

- Macro instructions
- Micro character strings
- Conditional sequences
- Duplicated sequences

The first character must be one of the following:

- Alphabetic character (A through Z or a through z)
- Dollar sign ($)
- Percent sign (%)
- At sign (@)

Characters 2 through 8 can also be decimal digits (0 through 9).

Names that are 1 to 255 characters in length can be used to identify the following types of information:

- Program modules
- Sections

The first character must be one of the valid name characters or the underscore (_) character. Characters 2 through 255 can also be decimal digits (0 through 9).

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 not valid names:

Valid	Comment
`count`	Lowercase is permitted
`@ADD`	`@` legal beginning character
`_SUBTRACT`	_ beginning character and 9 characters are legal
`ABCDE465`	Combinations of letters and digits are legal if the first character is legal

Not valid	Comment
`9knt`	Begins with a numeric character
`JOHNJONES`	Contains more than 8 characters
`Y+Z3`	Contains an illegal character
`+YZ3`	Begins with `+`

Note: UNICOS supports the Source Code Control System (SCCS) and UNICOS source manager (USM). If you plan to use SCCS or USM to store your CAL program, avoid using the 3-character string `%`*U*`%`; where *U* is any uppercase letter. SCCS and USM replace these strings throughout your source program with other text. Because this type of string is allowed within identifiers and long-identifiers, avoid using it in names, long names, and symbols.

The underscore character (_) also is used as the concatenation character by CAL (see subsection 3.3, page 41). Usually the assembler edits this character out of a source line. To insert this character into a long name, either disable editing or use the predefined concatenation micro (`$CNC`). To disable editing, use either the invocation statement or the `EDIT` pseudo instruction.

Symbols
4.3

A *symbol* is an identifier that can be from 1 to 255 characters long and has an associated value and attributes. You can use symbols in expressions and in the following ways:

- In the location field of a source statement to define the symbol for use in the program assign it a value and certain characteristics called attributes.
- In the operand or result field of a source statement to reference the symbol.
- In loader linkage

A symbol can be local or global depending on where the symbol is defined; that is, a symbol used within a single program module is local and a symbol used by a number of program segments is global (see subsection 3.1.2, page 35). A symbol also can be unique to a code sequence (see subsection 4.3.1.2, page 71).

CAL generates symbols of the following form (where *n* is a decimal digit):

`%%`*nnnnnn*

Symbols that begin with the character sequence `%%` are discarded at the end of a program segment regardless of whether they are redefinable or defined in the global definitions part, and regardless of whether they are user-defined or generated by CAL.

For more detailed information about symbols generated by CAL, see the description of the `LOCAL` pseudo instruction in subsection 6.11, page 184.

If a symbol is properly identified and defined as one of the registers reserved by CAL, a warning message is issued.

Symbols can be used if they are:

- Specified as unqualified or qualified.
- Defined or associated with a value and attributes.
- Assigned address, relative, and redefinable attributes.
- Referenced by using the value rather than the symbol itself.

Symbol qualification 4.3.1

Symbols defined within a program module (between `IDENT` and `END` pseudo instructions) can be unqualified or qualified. They are unqualified unless preceded by the `QUAL` pseudo instruction (see the `QUAL` pseudo instruction in appendix A, page 189, for more information).

Unqualified symbol 4.3.1.1

The following statements describe ways in which unqualified symbols can be referenced:

- Unqualified symbols defined in an unqualified code sequence can be referenced without qualification from within that sequence.
- If the symbol has not been redefined within the current qualifier, unqualified symbols can be referenced without qualification from within the current qualifier.
- Unqualified symbols can be referenced from within the current qualifier by using the form `//`*symbol.*

Unqualified symbols are defined as follows:

```
symbol  =   n   ; symbol is equal to n
```

The following example illustrates unqualified symbol definition:

```
       EDIT   OFF
       IDENT  TEST
SYM_1  =      *          ; SYM_1 has a value equal to the location
                         ; counter.
       A1     SYM_1      ; Register A1 gets SYM_1's value.
SYM_2  SET    2          ; SYM_2 is redefinable
SYM_3  =      3          ; SYM_3 is not redefinable.
```

Qualified symbols 4.3.1.2

You can make a symbol that is not a global symbol unique to a code sequence by specifying a symbol qualifier that will be appended to all symbols defined within the sequence. The `QUAL` pseudo instruction qualifies symbols (see the `QUAL` pseudo instruction in appendix A, page 189).

Qualified symbols must be defined with respect to the following rules:

- A qualified symbol cannot be defined with a label that is reserved for registers.
- Symbols can be qualified only in a program module.

Qualified symbols can be referenced as follows:

- If a qualified symbol defined in a code sequence is referenced from within that 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*. The *qualifier* variable is a 1- to 8-character identifier defined by the `QUAL` pseudo instruction and the *symbol* variable is a 1- to 255-character identifier.

Qualified symbols are defined as follows:

qualified_symbol = /[*identifier*]/*symbol*

The following example illustrates the use of qualified symbols:

```
      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
```

Symbol definition
4.3.2

A symbol is defined by assigning it a value and attributes. The value and attributes of a symbol depend on how the program uses the symbol. 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:
  - It has the address of the current value of the location counter (for a description of counters, see subsection 3.6.3, page 56).
  - It has parcel-address or word-address attributes.
  - It is absolute, immobile, or relocatable.
  - It is 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.

The following are examples of a symbol:

```
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.
```

Symbol attributes 4.3.3

When a symbol is defined, it assumes two or more attributes. These attributes are in three categories as follows:

- Address
- Relative
- Redefinable

Every symbol is assigned one attribute from each of the first two categories. Whether a symbol is assigned the redefinable attribute depends on how the symbol is used. Each symbol has a value of up to 64 bits associated with it.

Address attributes 4.3.3.1

Each symbol is assigned one of the following address attributes:

- 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 it is equated to an expression having a word-address attribute, or if word is explicitly stated in the operand field of an `EXT` pseudo instruction.

- 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, if it is equated to an expression having a parcel-address attribute, or if parcel is explicitly stated in the operand field of an `EXT` pseudo instruction.

- 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 instruction. All globally defined symbols have an address attribute of value.

- 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` or if it is equated to an expression that is absolute. All globally defined symbols have a relative attribute of absolute. The symbol is known only at assembly time.

Relative attributes 4.3.3.2

Each symbol is assigned one of the following relative attributes:

- 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` or if it is equated to an expression that is immobile. The symbol is known only 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 also is 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. The address attribute of an external symbol is specified 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.

  Note: The assignment of an unknown variable with a register at assembly time is made by using a symbol with a relative attribute of external.

In the following example, register `s1` is loaded with variable `ext1` at assembly time:

```
       ext    test1        ; 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
```

Redefinable attributes
4.3.3.3

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 various 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.

The following example illustrates the redefinable attribute:

```
       IDENT  TEST
SYM1   =      1               ; Not redefinable
SYM2   SET    2               ; Redefinable
SYM1   SET    2               ; Error: SYM1 previously defined as 1
SYM2   SET    3               ; Redefinable
       END
```

Symbol reference
4.3.4

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 can be found in the operand fields of pseudo instructions.

A symbol reference within an expression can contain a prefix that causes the value and attributes associated with the symbol to be altered. The prefix affects only the specific reference in which it occurs. For details, see subsection 4.6, page 90.

The following example illustrates a symbol reference:

```
        S1     SYM1+1       ; Register S1 gets the value 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.
```

Data
4.4

Some instructions manipulate data. CAL instructions use data of the following types:

- Constants
- Data items
- Literals

The subsections that follow describe these types of data.

Constants
4.4.1

Constants can be defined as floating, integer, or character.

Floating constant
4.4.1.1

A *floating constant* is evaluated as a one- or two-word quantity, depending on the precision specified. (See the floating-point data format figures in the appropriate symbolic machine instruction manual.)

The floating constant is defined as follows:

```
[ decimal-prefix ] floating-decimal [ binary-scale decimal-integer]
```

In the preceding definition, variables are defined as follows:

- *decimal-prefix*

  This variable specifies the numeric base for the *floating-decimal* and/or the *decimal-integer* variables. `D'` or `d'` specifies a *decimal-prefix* and is the only prefix available for a floating constant.

- *floating-decimal*

  The *floating-decimal* variable can include the *decimal-integer*, *decimal-fraction* and/or *decimal-exponent* variables. A *decimal-integer* is a nonempty string of decimal digits. A *decimal-integer* or a *decimal-fraction* is a nonempty string of decimal digits representing a whole number, a mixed number, or a fraction.

  A *floating-decimal* can be defined as follows:

  – 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 period (.) 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* or *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*

- *decimal-exponent*

  The power of 10 by which the integer and/or *fraction* will be multiplied; indicates whether the constant will 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* will 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

Note: Double-precision floating-point numbers are truncated to single-precision floating-point numbers if pseudo instructions, which can reserve only one memory word (such as the `CON` pseudo instruction) are used.

The following examples illustrate floating constants:

```
       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 nD+n.
SYM    =      1777752d+10    ; Double-precision floating constant of
                             ; the form nD+n.
```

Integer constant
4.4.1.2

An *integer constant* is evaluated as a 64-bit twos complement integer. The integer constant is defined as follows:

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*]

In the preceding definition, variables are defined as follows:

- *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 that will 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*, *octal-prefix*, *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 hexadecimal 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

The following examples illustrate integer constants:

```
       S1     O'1234567     ; Octal-prefix followed by octal-integer.
       A4     D'50          ; Integer-constant of the form
                            ; decimal-prefix followed by
                            ; decimal-integer.
SYM    =      x'ffffffa     ; Integer-constant of the form hex-prefix
                            ; followed by hex-integer.
```

Character constants 4.4.1.3

The character constant is defined as follows:

[*character-prefix*] *character-string* [*character-suffix*]

In the preceding definition, variables are defined as follows:

- *character-prefix*

  The character set used for the stored constant:

`A` or `a`	ASCII character set (default)
`C` or `c`	Control Data display code
`E` or `e`	EBCDIC character set

- *character-string*

  The default is a string of zero or more characters (enclosed in apostrophes) from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate one apostrophe.

- *character-suffix*

  The justification and fill of a 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

The following examples illustrate character constants:

```
        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).
```

Data items 4.4.2

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. Data items can be floating, integer, or character. The subsections that follow describe these types of data items.

Floating data item 4.4.2.1

Single-precision floating data items occupy one word and double-precision floating data items occupy two words. A floating data item is defined as follows:

[*sign*] *floating-constant*

In the preceding definition, the *sign* variable is defined as follows:

- *sign*

  The sign variable determines how the floating data item will be stored. The sign variable can be specified as follows:

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

Note: Although syntactically correct, # is not permitted; a semantic error is generated with floating data.

- *floating-constant*

  The syntax for a floating data item is the same as the syntax for floating constants. Floating constants are described in subsection 4.4.1.1, page 76.

The following example illustrates floating constants for data items:

```
        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   4.5E+1O          ; Single-precision floating constant of
                                ; the form n.nE+n.
        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.
```

Integer data item 4.4.2.2

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

[*sign*] *integer-constant*

In the preceding definition the *sign* variable defines the form of a data item to be stored. The *sign* variable can be replaced in the integer data item definition with any of the following:

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

The syntax for *integer-constant* is described in subsection 4.4.1.2, page 79.

The following example illustrates integer constants for data:

```
        DATA   +o'20           ; Octal integer
        VWD    40/0,24/O'200
```

Character data item 4.4.2.3

The character data item is as follows:

[*character-prefix*] *character-string* [*character-count*] [*character suffix*]

In the preceding definition, variables are defined as follows:

- *character-prefix*

  This variable specifies the character set used for the stored constant. It is specified as follows:

`A` or `a`	ASCII character set (default)
`C` or `c`	Control Data display code
`E` or `e`	EBCDIC character set

- *character-string*

  The default is a string of zero or more characters (enclosed in apostrophes) from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate one apostrophe.

- *character-count*

  The length of the field, in number of characters, into which the data item will 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; therefore, length is not necessarily an integral number of words. The character width is 8 bits for ASCII or EBCDIC, and 6 bits for control data display code.

If an asterisk is in the count field, the actual number of characters in the string is used as the count. Two apostrophes that are used to represent one apostrophe are counted as one character.

If the base is mixed, CAL assumes that the count is decimal. See appendix A, page, 189 for information on the `base` pseudo instructions.

- *character-suffix*

  This variable specifies justification and fill of the character string as follows:

`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

The following example illustrates character data items:

```
      DATA   A'ERROR IN DSN'   ; ASCII character set left justified and
                               ; blank fill by default; 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.
```

Literals
4.4.3

Literals are read-only data items whose storage is controlled by CAL. Specifying a literal lets you implicitly insert a constant value into memory. The actual storage of the literal value is the responsibility of the assembler. Literals can be used only 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 block of memory known as the *literals section*. Subsequent references to a literal value do not produce multiple copies of the same literal.

Because literals can map into the same location in the literals section, CAL checks for the presence of matching literals before new entries are added. This check is made bit by bit. If the current string is identical to any string currently stored in the literals section, CAL maps that string to the location of the matching string. If the current string is not identical to any of the strings currently stored, 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) to a constant section
  - 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 0. In actual use, you must specify at least one 6-bit character for the CDC character set or one 8-bit character for the ASCII (default) and EDCDIC character sets.
- By default, literals always fall on full-word boundaries. Trailing blanks are added to fill the word to the next word boundary.

When used as an element of an expression, a literal is defined as follows:

```
=data-item
```

A data item for literals is the same as data items for constants. Data items for constants are described in subsection 4.4.2, page 82.

Single-precision literals are stored in one 64-bit word (default). Double-precision literals are stored in two 64-bit words. The following example shows how literals can be specified with single or double precision:

```
        CON     =1.5          ; Single-precision literal
        CON     =1.sD1        ; Double-precision literal
```

Figure 6 illustrates how the ASCII character `a` is stored by either of the following instructions (^ represents a blank character):

```
CON   =‘a’H
CON   =‘a’
```

Figure 6. ASCII character with left-justification and blank-fill

Figure 7 illustrates how the ASCII character `a` is stored by any of the following instructions (^ represents a blank character):

```
CON   =‘a’L
CON   ‘a’R
CON   –‘a’S
```

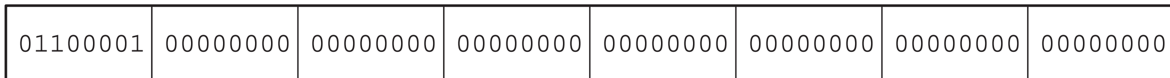


Figure 7. ASCII character with left-justification and zero-fill

Figure 8 illustrates how the ASCII character `a` is stored by the following instruction (^ represents a blank character):

```
CON   =‘a’R
```

This example illustrates how the ASCII character `a` is stored when `=‘a’R` is specified.

00000000	00000000	00000000	00000000	00000000	00000000	00000000	01100001

Figure 8. ASCII character with right-justification and zero-fill

Figure 9 illustrates how the ASCII character `a` is stored by the following instruction (^ represents a blank character):

```
CON   =‘a’*R
```

01100001	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure 9. ASCII character with right-justification in 8 bits

The three character sets available to CAL are declared as follows:

```
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.
```

The following example illustrates the use of the `H`, `L`, `R`, or `Z` options when specifying literals:

```
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
```

Special elements
4.5

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. For a description of expression elements, see subsection 4.7, page 94. The origin, location, word-bit-position, and parcel-bit-position counters are described in section 3, page 33.

Elements that have special meanings to the assembler are described as follows:

- `*`. Location counter.

  The asterisk (`*`) 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 the `LOC` pseudo instruction modified it by using an expression that has a relative attribute of absolute. The location counter is immobile if it is relative to either a `STACK` or `TASKCOM` section. The location counter is relocatable in all other cases.

- `*A` or `*a`. Absolute location counter.

  The `*A` or `*a` denotes a value equal to the current value of the location counter with parcel-address and absolute attributes.

- `*B` or `*b`. Absolute origin counter.

  The `*B` or `*b` 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.

- `*O` or `*o`. Origin counter.

  The `*O` or `*o` 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.

- `*W` or `*w`. Word pointer.

  The `*W` or `*w` 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.

  The `*P` or `*p` 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 it is almost always 0:

  ```
  SYM1 = *P
  ```

Element prefixes

4.6

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 parcel-address or word-address attributes, respectively, in the expression in which the reference appears.

A prefix does not permanently alter the attribute of a symbol. A prefix only effects the current reference.

Parcel-address prefix 4.6.1

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 4. Each Cray word is divided into 4 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 being addressed, multiply the word address by 4. 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.` specified for an element with value-address attribute does not cause the value to be multiplied by 4; however, the `P.`or `p.` prefix can be used to assign the parcel-address attribute to the element.

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

Figure 10, page 92, shows the octal 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 10. Word/parcel conversion for 6 words

The following example illustrates the use of the parcel-address prefix:

```
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.
```

Word-address prefix
4.6.2

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 4. 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	0	No

A `W.` or `w.` prefix specified for an element with a value-address attribute does not cause the value to be divided by 4. However, the `W.` or `w.` prefix can 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.

The following example illustrates the use of a word-address prefix:

```
SYM2   =     W.*          ; Word and relocatable attributes.
       A0    W.ADDR
       A4    W.BUFF+0'100
```

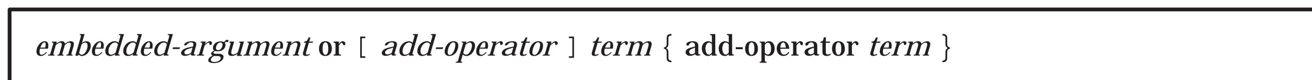
Expressions
4.7

The result and operand fields for many source statements contain expressions. An *expression* consists of one or more terms joined by special characters referred to as adding operators (*add-operator*). A *term* consists of one or more special elements, constants, symbols, or literals (*prefixed-element*) joined by multiplying operators (*multiply-operator*). Figure 11 diagrams an expression and Figure 12 diagrams a term.

Figure 11. Diagram of an expression

Figure 12. Diagram of a term

An expression is defined as follows:



The variables listed in the previous definition are defined in the subsections that follow.

Add-operator
4.7.1

An *add-operator* joins two terms in an expression or precedes the first term of an expression. *Add-operators* include the plus sign (+) and the minus sign (–) and perform addition and subtraction.

Terms
4.7.2

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.

A *term* is defined as follows:

prefixed-element { *multiply-operator prefixed-element* }

The following general rules apply:

- Only one *prefixed-element* within a term can have a relative attribute of immobile or relocatable. All other *prefixed-elements* in that term must have relative attributes of absolute.
- A *prefixed-element* with a relative attribute of external must be the only *prefixed-element* of the term. If preceded by an *add-operator*, that operator must be a `+`.
- The *prefixed-element* to the right of a slash (`/`) must have a relative attribute of absolute.
- A term that contains a slash (`/`) must have an attribute of absolute up to the point at which the `/` is encountered (see subsection 4.7.2.3, page 97).
- Division by 0 produces an error.

The following example illustrates the use of terms:

```
SYM    =    *            ; Relocatable and parcel attributes.
       S1   SYM1         ; 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.
```

The following are examples of terms:

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

Prefixed-elements 4.7.2.1

A *prefixed-element* is defined as follows:

> `[ # ] [` *element-prefix* `]` *element*

The variables in the previous definition are defined as follows:

- *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 attribute
`W. or w.`	Word-address attributes

  For more information about *element-prefixes*, see subsection 4.6, page 90.

- *element*

  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.`). For more information about *element*, see subsection 4.5, page 89.

The following are examples of elements:

`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 the asterisk (*) which specifies multiplication and the slash (/) which specifies division.

4.7.2.3 *Term attributes*

Each *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 final attribute of the term is the attribute in effect when the final (rightmost) 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* that compose a term) determine the relative attributes of the term.

CAL assigns every term a relative attribute determined by the following rules:

- A term assumes the attributes of *absolute* if every *prefixed-element* is absolute. For example:

  ```
  2*4/3*4
  ```

  In the above example, 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.

- 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. For example:

  ```
  STKSYM*3
  ```

  In the above example, immobile (STKSYM) * absolute (3) is evaluated as immobile.

- 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.

  ```
  2*SYM1*2
  ```

  In the above example, absolute (2) * relocatable (SYM1) is evaluated as relocatable. Relocatable (2*SYM1) * absolute (2) is evaluated as relocatable.

- A term assumes the attribute of *external* if it consists of one *prefixed-element* and the *prefixed-element* is external. For example:

  ```
  EXT1
  ```

  In the above example, one external (EXT1) element is evaluated as external.

  ```
  EXT2*SYM1
  ```

  In the above example, external (EXT2) * relocatable (SYM1) produces an error.

  In the following example, Absolute (4) * relocatable (SYM1) is evaluated as relocatable; relocatable (4*SYM1) / 4 produces an error:

  ```
  4*SYM1/4
  ```

  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, page 95.

CAL assigns one of the following address attributes to every term:

- Parcel-address
- Word-address
- Value

Figure 13, page 100, indicates how address attributes are assigned to terms and partial terms. *Vterm*, *Pterm*, and *Wterm* denote the attribute of the partial term resulting from all elements evaluated before the current element. In Figure 13, `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 the 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 2-digit value following an address attribute indicates that although a result is specified, a warning message is issued that corresponds to the 2-digit superscript. For example, if the partial term *interm* is divided by a *prefixed-element* with an address attribute of parcel, the result is value and message 84 is issued as follows:

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

See appendix B, page 281, for the text associated with messages.

* / (Partial term \ 2nd term)	V	P	W
Vterm	V / V	P / V84	W / V86
Pterm	P / P	P80 / V	V82 / V87
Wterm	W / W	V81 / V85	V83 / V

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

Figure 13. Address attribute assignment chart

Expression evaluation
4.8

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:

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*. See the following examples.

```
sym1   =    1           ; Valid expression
sym2   =    (1)         ; Syntactically correct, but CAL issues
                        ; error message.
```

An *embedded-argument* can be any *argument-character* that is enclosed in parentheses. For example:

```
       MACRO
       FRED   p1,p2
ABC    =      p1
       S1     ABC*p2
FRED   ENDM
       FRED   (1+2),3    ; (1+2) is the embedded argument
```

When a complete term is 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. Elements are complemented in the rightmost bits of a field width equal to the destination field.

Note: CAL processes *floating-constants* as expected when they are specified as one uncomplemented *prefixed-element* 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:

```
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 by using integer arithmetic:

```
G       CON    1.0+2.0
H       CON    -1*3.4
I       CON    -#1.0
```

This example demonstrates how the result of a `VWD` with a 9-bit destination field is stored; ^ represents a blank space.

```
        VWD    D'9/'abc'+1   ; The terms of the expression 'abc' and 1
```

Figure 14 and Figure 15 contain the binary representations of the ASCII character strings `"abc"` and `1`, respectively.

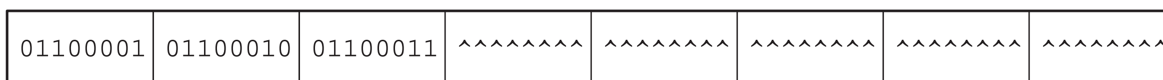


Figure 14. 64-bit binary representation of ASCII abc, left-justified

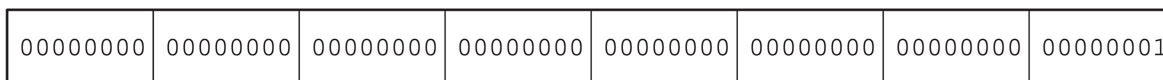


Figure 15. 64-bit binary representation of 1

Because 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 in Figure 16.

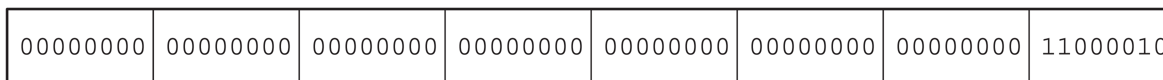


Figure 16. Binary representation of ASCII abc, right-justified in 9 bits

CAL adds the value `1` (Figure 15, page 102) to the value shown in Figure 16, page 102 (`011000010`), and stores it in the destination field (see Figure 17). CAL issues a warning message stating that the character string "`abc`" has been truncated. The destination field contains a value of 303 (`011000011`).

011000011

Figure 17. Result of `VWD` with 9-bit destination field

The following example demonstrates that elements are complemented in the rightmost bits of a field width equal to the destination field:

```
        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.
```

Figure 18 and Figure 19 contain the complement of `1` and the binary representation of the value 1 (0001), respectively.

11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111110

Figure 18. 64-bit binary representation of the complement of 1

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000001

Figure 19. 64-bit binary representation of 1

Figure 20 shows that the actual value of the complement of 1 is stored in the rightmost bits of a word in memory.

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00001110

Figure 20. Binary representation of the complement of 1 stored in the rightmost bits of a 4-bit field

The binary value 1110 (Figure 20) is stored in the destination field, and CAL adds the value 1 to the destination field; the result (1111) is shown as the rightmost 4 bits as in Figure 20 and is stored as shown in Figure 21.

1111

Figure 21. Result of `VWD` with 4-bit destination field

Evaluating immobile and relocatable terms with coefficients 4.8.1

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 an associated 64-bit integer coefficient 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 an associated 64-bit integer coefficient 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.

Each section has two relative section coefficients, one represents an immobile relative attribute and one represents a relocatable relative attribute. These relative section coefficients are initialized to 0 before the evaluation of each expression. As each term is evaluated within an expression, the coefficient of the term 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 0.
- Immobile; if the expression contains no external terms and all of the coefficients for all of the sections are 0, except for one immobile coefficient that must have a value of 1. The expression is immobile relative to the section with the coefficient of 1.
- Relocatable; if the expression contains no external terms and all of the coefficients for all of the sections are 0 except for one relocatable coefficient that must have a value of 1. The expression is relocatable relative to the section with the coefficient of 1.
- External; if the expression contains one external term and all of the coefficients for all of the sections are 0.
- Not valid; all other cases.

For example, if `SYMBOL` is assumed to be relocatable, `SYMBOL*2+1–SYMBOL` is considered a valid expression when evaluated by CAL. Because `SYMBOL` is relocatable, substituting 1 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 affect the evaluation of the relocatable coefficient. The third term (`–1`) includes the relocatable term `SYMBOL`. A 1 is subtracted from the coefficient maintained by CAL for the relocatable section `SYMBOL`.

When the entire term is evaluated, the coefficient associated with the relocatable term `SYMBOL` equals 1. Because all relocatable terms within the expression are relative to one section and the final coefficient of the section is 1, the expression is relocatable relative to that section.

Every relocatable symbol is relative to a section. All sections contain an initial coefficient of 0 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; the binary that will be evaluated is 100*010+001–100:

```
SYMBOL*2+1–SYMBOL
```

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 you would expect to be generated. The following example demonstrates the correct and incorrect use of a relocatable term:

```
       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
```

In the preceding example, the term SYMBOL*2+1 is not relocatable because the results generated depend on the location of the module by the loader. 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 an error-level diagnostic message.

The following example illustrates the use of relocatable terms:

```
         IDENT          TEST
SNAME1   SECTION
SYMBOL1  BSS            4
SYMBOL2  =              w.*
         BSS            5
SNAME2   SECTION
SYMBOL3  BSS            3
         SECTION        *
SYMBOL4  =              3*SYMBOL2+SYMBOL3-1SYMBOL2-2*SYMBOL1
         END
```

In the previous example, the expression `3*SYMBOL2+SYMBOL3-1-SYMBOL2-2*SYMBOL1` contains five terms, four of which are relocatable; it 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`)

The coefficients for the `SNAME1` and `SNAME2` sections were initialized to 0 before the expression was evaluated. The main section has a coefficient of 0. When the coefficients for the relocatable terms relative to `SNAME1` are evaluated, the result is `0` (`+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 13, page 100)
+SYMBOL3	Word	Word (see Figure 13, page 100)
-1	Value	Value (see Figure 13, page 100)
-SYMBOL2	Word	Word (see Figure 13, page 100)
2*SYMBOL1	Value*word	Word (see Figure 13, page 100)

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, page 110.

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

Term	Result	Description
3*SYMBOL2	3*4=0'14	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.
-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 (3*SYMBOL2+SYMBOL3-1-SYMBOL2-2*SYMBOL1) expression, the result is 7.

The following example illustrates the use of immobile terms:

```
         ident          test
taskc    section        taskcom
tcsym    bss            4
         section        *
symbol   =              taskc+tcsym-taskc
```

In the preceding example, the `taskc+tcsym-taskc` expression 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`)

The relative section coefficients for relocatable `taskc` and immobile `tcsym` were initialized to 0 before the expression was evaluated. When the coefficients for the relocatable terms relative to `taskc` are evaluated, the result (`+1-1=0`) is 0. When the coefficient for the immobile term (`tcsym`) is evaluated, the result (`+1`) is 1. Because 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 13, page 100)
`+tcsym`	Word	word (see Figure 13, page 100)
`-taskc`	Word	word (see Figure 13, page 100)

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, page 110.

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.

Expression attributes
4.9

To determine the expression attributes for a full expression, evaluate the terms within an expression. The assembler can assign the following attributes to an expression:

- Relative

  Relative attributes are classified as follows:
  - Absolute
  - Immobile
  - Relocatable
  - External

- Address

  Address attributes are classified as follows:
  - Parcel-address
  - Word-address
  - Value

Relative attributes 4.9.1

Every expression assumes one of the relative attributes as follows:

- 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 is 0 for each section within the current module represented in the expression. The exception is when one section has a coefficient of +1 (positive relocation) and is immobile with respect to that expression.
- An expression is *relocatable* if the coefficient for every section within the current module represented in the expression is 0. The exception is when one section has a coefficient of +1 (positive relocation) and is relocatably associated with that expression. 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 it 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 instruction have a value of 0 associated with them.

The following are examples of relative attributes (see section 3, page 33, for a description of sections):

```
       IDENT  TEST
       EXT    EXT1
SNAME1 SECTION
SYM1   BSS    4
SYM2   =      W.*
       BSS    5
       SYM4   EXT1+SYM1       ; Illegal external term and relocatable
                              ; terms with coefficients of 1 in the
                              ; same expression.
       SYM5   EXTl+SYMl-SYM2  ; Legal; SYMl (+1) and SYM2 (-1) cancel
                              ; each other and produce a coefficient
                              ; of 0 for the expression.  The value of
                              ; the expression EXTl+SYMl-SYM2 is 4
                              ; (0+0-4).
       END
```

Address attributes
4.9.2

Each expression assumes an address attribute as follows:

- An expression has a *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 a *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.

Truncating expression values
4.9.3

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

The following example illustrates expression value truncation:

```
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 leftmost bits lost in truncation are not all 0's or all 1's with the leftmost remaining bit also 1 (that is, a negative quantity).

In the preceding example, truncation occurs in statement `VWD 5/-1` (see Figure 22), but an error message is not generated because the part that was truncated included all 1's and the leftmost bit of the 5-bit field is also a 1.

11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
Truncated							Result

Figure 22. 64-bit binary representation of –1

11111

Figure 23. Truncated value of –1 stored in a 5-bit field

Truncation occurs in statement `VWD 3/5`. An error message is not generated, because the truncated part was all 0's. The result is truncated and stored as shown in Figure 24.

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000101
Truncated							

Figure 24. 64-bit binary representation of 5

101

Figure 25. Truncated value of 5 stored in a 3-bit field

Truncation occurs in statement `VWD 2/5`. CAL generates a warning message, because a combination of 1's and 0's is truncated. The result is truncated and stored as shown in Figure 26.

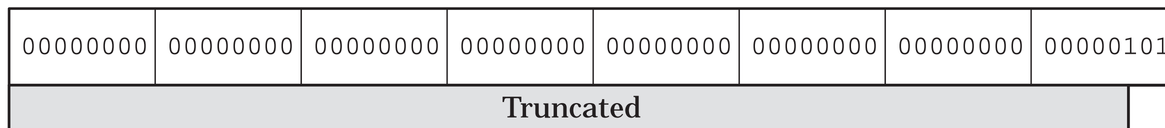


Figure 26. 64-bit binary representation of 5

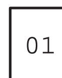


Figure 27. Truncated value of 5 stored in a 2-bit field

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

If a message of error-level is issued for an expression it causes the expression to have a relative attribute of absolute, an address attribute of value, and a value of 0.

The following are examples of expressions:

Expression	Description
ALPHA	An expression consisting of one 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-*`	All relocation of terms 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`.
`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.

Pseudo Instructions [5]

Pseudo instructions direct the assembler in its task of interpreting source statements and generating an object program.

> **Note:** A detailed description of the pseudo instructions presented in this section are listed in alphabetical order in appendix A, page 189.

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

Definitions of symbols, micros, macros, or opdefs included before the first `IDENT` pseudo instruction or between an `END` and a subsequent `IDENT` pseudo instruction are global and can be referenced in any subsequent program module (see subsection 3.1.2, page 35).

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

Symbolic machine instructions and the following pseudo instructions 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 immediately after a macro or opdef prototype statement or after a `DUP` or `ECHO` pseudo instruction. Comment statements can intervene. All other pseudo instructions, macro definitions, and opdef definitions can appear anywhere in a CAL program.

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, STACK, ORG, LOC, BITW, BITP, BSS, ALIGN
Message control	ERROR, ERRIF, MLEVEL, DMSG
Listing control	LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, ENDTEXT
Symbol definition	=, SET, MICSIZE, DBSM
Data definition	CON, BSSZ, DATA, VWD
Conditional assembly	IFA, IFC, IFE, IFM, SKIP, ENDIF, ELSE
Micro definition	CMICRO, MICRO, OCTMIC, DECMIC
File control	INCLUDE
Defined sequences	MACRO, OPDEF, DUP, ECHO, ENDM, ENDDUP, STOPDUP, LOCAL, OPSYN, EXITM, NEXTDUP

Note: You can specify pseudo instructions in uppercase or lowercase, but not in mixed case.

Program control
5.1

The program control pseudo instructions define the limits of a program module and include the following:

Pseudo	Description
`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.

Loader linkage
5.2

The loader linkage pseudo instructions provide for the loading of multiple object program modules, linking them into one executable program (`ENTRY` and `EXT`), and specifying the main program entry (`START`).

The loader linkage pseudo instructions include the following:

Pseudo	Description
`ENTRY`	Specifies symbols, defined as addresses or values, so that 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 at which execution begins.

Mode control
5.3

Mode control pseudo instructions define the characteristics of an assembly. The `BASE` pseudo instruction 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 the editing of assembler statements. The `FORMAT` pseudo instruction controls the format that is used for interpreting assembly source statements.

The mode control pseudo instructions include the following:

Pseudo	Description
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.

Section control
5.4

Section control pseudo instructions control the use of sections and counters in a CAL program.

The section control pseudo instructions include the following:

Pseudo	Description
SECTION	Defines specific program sections and replaces the BLOCK and COMMON pseudo instructions. The SECTION pseudo instruction is recommended for use with all Cray PVP systems because it includes all of the capabilities of BLOCK and COMMON pseudo instructions.
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 origin counters.
LOC	Resets location counter.
BITW	Sets the current bit position relative to the current word.
BITP	Sets the current bit position relative to the current parcel.

Pseudo	Description
BSS	Reserves memory.
ALIGN	Aligns code on an instruction buffer boundary.

Message control
5.5

Two pseudo instructions, ERROR and ERRIF, let you generate an assembly error condition. The MLEVEL pseudo instruction lets you change the level of messages you receive in your source program.

Pseudo	Description
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
DMSG	Issues a comment-level message containing the string found in the operand field

Listing control
5.6

Listing control pseudo instructions control the content and format of the listing produced by the assembler. These pseudo instructions are not listed unless the LIST pseudo instruction is specified by using the LIS option.

The listing control pseudo instructions are as follows:

Pseudo	Description
LIST	Controls listing by specifying particular listing features that will be enabled or disabled
SPACE	Inserts blank lines in listing
EJECT	Begins new page
TITLE	Prints main title on each page of listing
SUBTITLE	Prints subtitle on each page of listing

Pseudo	Description
`TEXT`	Declares beginning of global text source
`ENDTEXT`	Terminates global text source

Symbol definition
5.7

The `=`, `SET`, and `MICSIZE` pseudo instructions define symbols used in the program. Requirements for symbols are specified in subsection 4.3, page 69. The symbol definition pseudo instructions are as follows:

Pseudo	Description
`=`	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.
`DBSM`	Generates a named label entry in the debug symbol tables with a specific type specified.

Data definition
5.8

Data definition pseudo 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 both.

The data definition pseudo instructions are as follows:

Pseudo	Description
`CON`	Places an expression value into one or more words
`BSSZ`	Generates words that have been initialized to 0
`DATA`	Generates one or more words of numeric or character data

Pseudo	Description
VWD	Generates a variable-width field of word-oriented data

Conditional assembly
5.9

The conditional assembly pseudo instructions permit the optional assembly or skipping of source code. The conditional pseudo instructions IFA, IFC, or IFE determine whether the sequence of instructions following the test will be skipped or assembled. The end of the conditional sequence is determined by a count of instructions provided in 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 the statements that follow it.

When skipping under the control of a statement count, comment statements (denoted by an asterisk (*) in column 1) and continued lines are not included in the statement count.

When an IFA, IFE, IFC, SKIP, or ELSE pseudo instruction initiates skipping, editing is disabled. When the skip sequence is completed, the assembler returns to the editing mode in effect before skipping was initiated.

To specify a conditional assembly, use the following pseudo instructions:

Pseudo	Description
IFA	Tests expression attributes; address and relative attributes.
IFE	Tests two expressions for some assembly condition; less than, greater than, and equal to.
IFC	Tests two character strings for assembly condition; less than, greater than, and equal to.
IFM	Test for machine characteristics.
SKIP	Unconditionally skip subsequent statements.

Pseudo	Description
`ENDIF`	Terminates conditional code sequence.
`ELSE`	Reverses assembly condition.

Micro definition
5.10

Through the use of micros, programmers 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 following pseudo instructions specify micro definition:

Pseudo	Description
`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 predefined micros. They can be specified in all uppercase or all lowercase, but not mixed case. CAL provides the following predefined micros:

Micro	Description
`$DATE`	Current date – '*mm/dd/yy*'
`$JDATE`	Julian date – '*yyddd*'
`$TIME`	Time of day – '*hh:mm:ss*'
`$MIC`	Micro character – double quotation mark (")

Micro	Description
$CNC	Concatenation character – underscore (_).
$QUAL	Name of qualifier in effect; if none, null string.
$CPU	Target machine: 'CRAY YMP', 'CRAY C90', 'CRAY J90', or 'CRAY TS'.
$CMNT	Comment character used with the new format – semicolon (;).
$APP	Append character used with the new format – circumflex (^).
AREGSIZE	Number of bits in an A register of the current target machine. For CRAY C90, CRAY J90, or CRAY Y-MP systems, AREGSIZE = 32. For CRAY T90 systems, AREGSIZE = 64.
PREGSIZE	Number of bits in the Program register of the current target machine. For CRAY J90 and CRAY Y-MP systems, PREGSIZE = 24. For CRAY C90 systems, PREGSIZE = 32. For CRAY T90 systems, PREGSIZE = 32

The following example illustrates the use of a predefined micro ($DATE):

```
       DATA    'THE DATE IS "$DATE"'
       DATA    'THE DATE IS 06/23/94'†
```

You can reference micro definitions anywhere in a source statement, except in a comment, by enclosing the micro name in quotation marks. If column 72 of a line is exceeded because of a micro substitution, the assembler creates additional continuation lines. No replacement occurs if the micro name is unknown or if one of the micro marks was omitted.

In the following example, a micro called PFX is defined as the character string ID. A reference to PFX is in the location field of a line.

```
"PFX"TAG   S0             S1    ; Left-shifted three spaces when edited.
```

In the following example, before the line is interpreted, CAL substitutes the definition for PFX producing the following line:

```
IDTAG    S0               S1    ; Left-shifted three spaces when edited.
```

The following example shows the use of the predefined micros, AREGSIZE and PREGSIZE:

```
A     =     "AREGSIZE"          ; Size of the A registers.
      CON   A                   ; Store value in memory.
B     =     "PREGSIZE"          ; Size of the Program register.
      CON   B                   ; Store value in memory.
```

File control
5.11

The file control psuedo instruction, INCLUDE, inserts a file at the current source position. The INCLUDE pseudo instruction always prepares the file for reading by opening it and positioning the pointer at the beginning.

You can use this pseudo instruction to include the same file more than once within a particular file.

You can also nest INCLUDE instructions. Because you cannot use INCLUDE recursively, you should review nested INCLUDE instructions for recursive calls to a file that you have already opened.

Defined Sequences [6]

Defined sequences are sequences of instructions that can be saved for assembly later in the source program. Defined sequences have several functional similarities.

The four types of defined sequences are specified by the `MACRO`, `OPDEF`, `DUP`, and `ECHO` pseudo instructions. The `ENDM`, `ENDDUP`, and `STOPDUP` pseudo instructions terminate defined sequences. The `LOCAL` and `OPSYN` pseudo instructions are associated with definitions and are included in this section.

The defined sequence pseudo instructions are as follows:

Pseudo	Description
`MACRO`	A sequence of source program instructions 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 saved by the assembler for inclusion in a program called for by the `OPDEF` pseudo instruction. The opdef resembles a symbolic machine instruction.
`DUP`	Introduces a sequence of code that is assembled repetitively a specified number of times; 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.

Pseudo	Description
`LOCAL`	Specifies unique strings that are usually used as symbols within a `MACRO`, `OPDEF`, `DUP`, or `ECHO` pseudo instruction.
`OPSYN`	Defines a location field functional that is the same as a specified operation in the operand field functional.
`EXITM`	Terminates the innermost nested `MACRO` or `OPDEF` expansion.

For more information on macros and opdefs see the *UNICOS Macros and Opdefs Reference Manual*, publication SR–2403.

Similarities among defined sequences
6.1

Defined sequences have the following functional similarities:

- Editing
- Definition format
- Formal parameters
- Instruction calls
- Interact with the `INCLUDE` pseudo instruction

Editing
6.1.1

Assembler editing is disabled at definition time. The body of the definition (see subsection 6.1.2, page 129) is saved before micros and concatenation marks are edited.

If editing is enabled, editing of the definition occurs during assembly each time it is called. The `ENDDUP`, `ENDM`, `END`, `INCLUDE`, and `LOCAL` pseudo instructions and prototype statements should not contain 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 all parameter substitutions marked at definition time
- Edits the statement if editing is enabled
- Processes the statement

By disabling editing at definition time (default) and specifying the `INCLUDE` pseudo instruction with embedded underscores, a saving in program overhead is achieved. Because editing is disabled at definition time, concatenation does not occur until the macro is called. If editing is enabled when the macro is called, the file is included at that time. This technique is demonstrated in the following example:

```
MACRO
INC
.
.
.
IN_CLUDE MYFILE     ; INCLUDE pseudo instruction with an embedded
.                   ; underscore
.
.
ENDM
```

Embedding underscores in an `INCLUDE` pseudo instruction becomes desirable when the `INCLUDE` pseudo instruction identifies large files. Because files are included when the macro is called and not at definition time, embedding underscores in the `INCLUDE` pseudo instruction can reduce the overhead required for a program.

Definition format 6.1.2

`MACRO`, `OPDEF`, `DUP`, and `ECHO` pseudo instructions 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 macro functional

definition 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 do not affect 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, page 44.

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 instruction. The body of a definition can be empty, or it can include other definitions and calls. A definition used within another definition is not recognized, however, 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 l 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.

Formal parameters
6.1.3

Formal parameters are defined in the definition header and recognized in the definition body. Four types of formal parameters are recognized as follows:

- Positional
- Keyword
- Echo
- Local

The characters that identify positional, keyword, echo, and local parameters must all have unique names within a given definition. Positional, keyword, and echo parameters are also case-sensitive. To be recognized, you must specify these parameters in the body of the definition exactly as specified in the definition header. Parameter names must meet the requirements for identifiers as described in subsection 4.2, page 67.

You can embed a formal parameter name within the definition body; however, embedded parameters must satisfy 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 `ABC_PARAM_DEF` string 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 an element separator (blank character):

  ```
  PARAM678
  ```

  In the following example, the embedded parameter is illegal because it is followed by the identifier-character `9`:

  ```
  PARAM6789
  ```

- Embedded parameters must contain 8 or less characters. `PARAM6789` is illegal because it contains 9 characters. The character that follows an embedded parameter (`9`) cannot be an identifier-character.
- If and only if the new format is specified, an embedded parameter must occur before the first comment character (`;`) of each statement within the body.
- 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` pseudo instructions. If any of these are used as parameter names, substitution of actual arguments occurs when these names are contained in any inner definition reference.

  Note: If the file is included at expansion time, arguments are not substituted for formal parameters into statements within included files.

Instruction calls
6.1.4

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 that is being assembled, another entry is made to the list of defined sequences, and assembly continues with the new definition sequence belonging to the inner, or nested, call.

At the end of a definition sequence, 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 rightmost character positions are overwritten.

If the sequence field (columns 73 through 90) of the source file is not empty, CAL copies the existing field for a call into every statement expanded by the call reserving columns 89 and 90 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 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) of the source file is empty, CAL inserts the name of the definition, as follows:

Name	Description
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 instruction, 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 asterisks (**) to indicate 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.

The following example illustrates how CAL tracks the nesting sequence:

```
*MCALL     :1
*MCALL     :2
*MCALL     :2
*MCALL     :2
*MCALL     :3
** MCALL   :*
** MCALL   :1
```

Interaction with the INCLUDE *pseudo instruction*
6.1.5

The INCLUDE pseudo instruction operates with defined sequences, as follows:

Sequence	Description
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 instruction statement itself is not included in the statements being counted.
ECHO	INCLUDE pseudo instructions are expanded at definition time.

Macros (MACRO)
6.2

A macro definition identifies a sequence of statements. This sequence of statements is saved by the assembler for inclusion elsewhere in a program. A macro is referenced later in the source program by the *macro call*. Each time the macro call occurs, the definition sequence is placed into the source program.

You can specify the MACRO pseudo instruction 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.

If a macro definition occurs within the global definitions part of a program segment, it is defined as global. If macro definitions occur within a program module (an IDENT, END sequence), they are local. A global definition can be redefined locally, however, at the end of the program module, it is reenabled and the local definition is discarded. A global definition can be referenced from anywhere within the assembler program following the definition.

The following example illustrates a macro definition:

```
       MACRO
       GLOBAL                 ; Globally defined.
* GLOBAL DEFINITION IS USED.
GLOBAL ENDM
       LIST   MAC
       GLOBAL                 ; Call to global definition.
* GLOBAL DEFINITION IS USED.
       IDENT  TEST
       GLOBAL                 ; Call to global definition.
* GLOBAL DEFINITION IS USED.
       MACRO                  ; Locally defined.
       GLOBAL                 ; Attempted global definition.
* Redefinition warning message is issued
* LOCAL DEFINITION IS USED.
GLOBAL ENDM
       GLOBAL                 ; Call to local definition.
* LOCAL DEFINITION IS USED.
       END                    ; Local definitions discarded
       IDENT  TEST2
       GLOBAL                 ; Call to global definition.
* GLOBAL DEFINITION IS USED.
       END
```

Macro definition
6.2.1

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 subsection 6.6, page 177.

The following macro definition transfers an integer from an A register to an S register and converts it to a normalized floating-point number:

```
          macro
          intconv          p1,p2 ; p1=A reg, p2=S reg.
          p2               +f_p1 ; Transfer with special exp and sign
                                 ; extension.
          p2               +f_p1 ; Normalize the S register.
intconv   endm                   ; End of macro definition.
```

As with every macro, INTCONV begins with the MACRO pseudo instruction. The second statement is the prototype statement, which names the macro and defines the parameters. The next three statements are definition statements that identify what the macro does. The ENDM pseudo instruction ends the macro definition.

The format of the macro definition is as follows:

```
ignored        MACRO        ignored
[location]     functional   parameters
               LOCAL        [name][,[name]]
               .
               .
               .
functional     ENDM
```

The variables in the above macro definition are described as follows:

- *location*

  The *location* variable specifies an optional location field parameter. It must be terminated by a space and it must meet the requirements for names as described in subsection 4.2, page 67.

- *functional*

  The *functional* variable specifies the name of the macro. It 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.

- *parameters*

  The *parameters* variable specifies *positional* and/or *keyword* parameters. Positional parameters must be entered before keyword parameters. Keyword parameters do not have to follow positional parameters. The syntax of the *parameter* variable is as follows:

```
positional-parameters[ , [keyword-parameters]]
```

The syntax for *positional-parameters* is described as follows:

```
[[!][*]name][,positional-parameters]
```

The variables that comprise the positional parameter are described as follows:

– `!/*`

The exclamation point (`!`) is optional. If it is not included, the *positional-parameter's* argument can be an embedded argument, character string, or null string. If the exclamation point (`!`) is included, the parameter can be a syntactically valid expression or a null string.

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 or pairs of matching left and right parentheses. If an asterisk precedes the positional parameter name, the embedded argument is used in its entirety. If an asterisk does not precede the positional parameter name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

A *character string* can be any character up to but not including a legal terminator (space, tab, or semicolon for new format) or an element separator (comma). If CAL finds an open parenthesis (character string) with no closing parenthesis (which would make it an embedded-argument), the following warning-level message is issued:

```
Embedded argument was not found.
```

A *syntactically valid expression* can include a legal terminator (space, tab, 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. If the syntactically valid expression is an embedded argument, then, as long as an asterisk precedes the *positional-parameter* name, the embedded argument is used in its entirety. If an asterisk does not precede the *positional-parameter* name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument. Use of the syntactically valid expression permits you to enter a string (`=',' R`) of characters that may contain one or more spaces or a comma.

The *null string* is an empty string.

– *positional-parameters*

positional-parameters must be specified with valid and unique names and they must meet the requirements for names as described in subsection 4.2, page 67. There can be none, one, or more positional 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 that are specified in the definition body must identically match positional parameters defined by the macro prototype statement.

The syntax for *keyword-parameters* can be any of the following:

```
[*]name=[expression-argument-value]
[,[keyword-parameters]]
[*] ! name=[expression-argument-value]
[,[keyword-parameters]]
[*]name=[string-argument-value]
[,[keyword-parameters]]
```

The elements of *keyword-parameters* are described as follows:

– *keyword-parameters*

  keyword-parameters must be specified with valid and unique names. Names within *keyword-parameter* must meet the requirements for names as described in subsection 4.2, page 67.

  There can be zero, one, or more *keyword-parameters*. You can enter names within *keyword-parameters* in any order. Default arguments can be provided for each *keyword-parameter* at definition time, and they 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 `-s` option argument can be an embedded argument, a character string, or a null string. If the `!` is included, the parameter be either a syntactically valid expression or a null string.

  Embedded argument. A left parenthesis signals the beginning of an embedded argument and it must be matched by a right parenthesis. An embedded argument can also contain pairs of matching left and right parentheses. If an asterisk precedes the positional parameter name, the embedded argument is used in its entirety; otherwise, 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, tab, or semicolon for new format) or an element separator. If CAL finds an open parenthesis (character string) with no closing parenthesis (which would make it an embedded argument), the following warning-level listing message is issued:

```
Embedded argument was not found.
```

  The null argument is an empty string.

Syntactically valid expression. An expression can include a legal terminator (space, tab, or semicolon for new format) or an element separator (comma). The syntactically valid expression is a legal expression, but it is used only as an argument and is not evaluated in the macro call itself.

If the syntactically valid expression is an embedded argument and if an asterisk precedes the positional parameter name, the embedded argument is used in its entirety. If an asterisk does not precede the *positional-parameter* name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

If a default is provided for a *keyword-parameter*, it must meet the preceding requirements.

The following example illustrates the use of positional parameters:

```
         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 precede the parameter on the prototype statement. The positional parameter `p2`, however, receives a value of (`v2`) because an asterisk precedes the parameter on the prototype statement.

```
       macro
       paren  p1,p2      ; Macro prototype.
       .
       .
       .
paren  endm
       paren  (v1),(v2)  ; Macro call.
```

Macro calls 6.2.2

An instruction of the following format can call a macro definition:

```
[locarg]    functional    positional-arguments ["," [keyword-arguments]]
[locarg]    functional    keyword-arguments
```

The elements of the macro call are described as follows:

- *locarg*

  The *locarg* element specifies an optional location field argument. *locarg* must be terminated by a space or a tab (new format only). *locarg* can be any character up to but not including a space. If a location field parameter is specified on the macro definition, you can specify a matching location field parameter 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*

  The *functional* element specifies the macro functional name. It must be an identifier or an equal sign. *functional* must match the functional specified in the macro definition.

- *positional-arguments*

  Positional-arguments specify an actual argument string that corresponds 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 are an actual argument string that corresponds to a *keyword-parameter* 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); *n* is the number of positional parameters in the operand field of the macro definition.

  You can list *keyword-arguments* in any order; matching the order in which *keyword-parameters* are listed on the macro prototype statement is unnecessary. However, because the *keyword-parameter* is case-sensitive, it 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*.

All arguments must meet the requirements of the corresponding parameters as specified in the macro definition prototype statement.

> **Note:** The `!` and `*` are not permitted on the macro call statement. These characters specified in the prototype statement for *positional-parameters* or *keyword-parameters* are remembered by CAL when the macro is called.

To call a macro, use its name in a code sequence. The INTCONV macro is called as follows:

```
       MACRO
       INTCONV            P1,P2 ; P1=A reg, P2=Sreg
       P2                 +F_P1 ; Transfer with special expression and
                                ; sign extension.
       P2                 +F_P2 ; Normalize the S register.
INTCO  ENDM                     ; End of macro definition.
       LIST               MAC
```

Call and expansion of the INTCONV macro:

```
     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. Because comments are not assembled, _* comments allow arguments to be shown without regard to hardware differences or available machine instructions.

The following examples show the use of *positional-parameters* and *keyword-parameters*.

The macro table contains positional and keyword parameters.

```
       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
```

The following shows the call and expansion of the `table` macro:

```
       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.
```

Macro `noorder` demonstrates that *keyword-parameters* are not order dependent.

```
       macro
       noorder           param1,param2,param3=,param4=b
       s1                param1
       s2                param2
       s3                param3
       s4                param4
noorderendm
       list              mac
```

The call and expansion of the `noorder` macro is as follows:

```
       noorder           (1),2,param4=dog,param3=d
       s1                1
       s2                2
       s3                d
       s4                dog
```

Macros ONE, two, and THREE demonstrate that the number of parameters specified in the macro call may form the number of parameters specified in the macro definition.

```
       MACRO
       ONE                  PARAM1,PARAM2,PARAM3
_*PARAMETER1:               PARAM1
                                 ; SYM1 corresponds to PARAM1.
_*PARAMETER2:               PARAM2
                                 ; Null string.
_*PARAMETER3:               PARAM3
                                 ; Null string.
ONE    ENDM
       LIST                 MAC
```

The call and expansion of the ONE macro using one parameter is as follows:

```
       ONE                  SYM1  ; Call using one parameter.
* PARAMETER 1:              SYM1  ; SYM1 corresponds to PARAM1.
* PARAMETER 2:                   ; Null string.
* PARAMETER 3:                   ; Null string.
       macro
       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
```

The call and expansion of the `two` macro using two parameters is as follows:

```
       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.
       MACRO
       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
```

The call and expansion of the `THREE` macro using prototype parameters is as follows:

```
       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.
```

The following examples demonstrate the use of the optional `!`.

Macro `BANG` demonstrates the use of the embedded argument (`1,2`), syntactically valid expressions for *positional-parameters* (`'abc,def'`), *keyword-parameters* (`PARAM3=1+2`), and the null string.

```
       MACRO
       BANG   PARAMl,!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
```

The call and expansion of the `BANG` macro is as follows:

```
       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.
```

In the previous example:

- If the argument for `PARAM1` had been (((`1,2`))), `S1` would have received ((`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.

In the following macro, called `remem`, the `!` is remembered from the macro definition when it is called:

```
       macro
       remem  !param1=' 'r    ; Prototype statement includes !
       s1     param1
remem  endm
       list   mac
```

The call and expansion of the `remem` macro is as follows:

```
       remem  param1=`,'r     ; Macro call does not include !
       s1      `,'r
```

The `NULL` and `nullparm` macros that follow demonstrate the effect of null strings when parameters are passed.

`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, which is the difference between the length of the parameter (`P2`) and the null string.

```
       MACRO
       NULL   P1,P2,P3
       S1     P1
       S2     P2              ; Left shifted two places.
       S3     P3
NULL   ENDM
       LIST   MAC
```

The call and expansion of the `NULL` macro is as follows:

```
       NULL   1,,3            ; Macro call.
       S1     1
*    S2                       ; Left shifted two places.
       S3     3
```

Macro `nullparm` demonstrates how a macro is expanded when the macro call does not include the location field name specified on the macro definition.

```
         macro
         nullparm         longparm
                                ; Prototype statement.
longparm =                1
nullparm endm
         list             mac
```

The call and expansion of the `nulparm` macro is as follows:

```
         nullparm
=        1
```

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 8 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.

The following macro, `DEFAULT`, illustrates how defaults are assigned for keywords when the macro is expanded:

```
         MACRO
         DEFAULT            PARAM1=(ABC DEF,GHI),PARAM2=ABC,PARAM3=
_* PARAM 1
_* PARAM 2
_* PARAM 3
DEFAULT  ENDM
         LIST               MAC
```

The following illustrates calls and expansions of the `DEFAULT` macro:

```
       DEFAULT          PARAM1=ARG1,PARAM2=ARG2,PARAM3=ARG3
                              ; Macro call.
*ARG1
*ARG2
*ARG3
       DEFAULT          PARAM1=,PARAM2=(ARG2),PARAM3=ARG3
*ARG2
*ARG3
       DEFAULT          PARAM1=((ARG1)),PARAM2=,PARAM3=ARG3
                              ; Macro call.
* (*ARG1)
* ARG3
```

The following examples illustrate the correct and incorrect way to specify a literal string in a macro definition.

Macro `WRONG` shows the incorrect 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.

```
       MACRO
       WRONG  PARAM1=' 'R      ; Prototype statement.
_* PARAM1
WRONG  ENDM                    ; End of macro definition.
       LIST   MAC              ; List expansion.
```

The call and expansion of `WRONG` is as follows (CAL erroneously expands `WRONG; ' 'R` was intended):

```
       WRONG                  ; Macro call
 * '
```

Macro `right` shows the correct way to specify a literal string in a macro definition.

```
       macro
       right  !param1=' 'r    ; Prototype statement.
 * param1
right  endm                   ; End of macro definition.
       list   mac             ; List expansion.
```

The expansion of right is as follows (CAL expands `right` as intended because of the !):

```
       right                  ; Macro call.
 * ' 'r
```

The following macros demonstrate the wrong and right methods for replacing parameters on the prototype statement with parameters on the macro call statement.

Macro `BAD` demonstrates the wrong method of replacing parameters.

```
       MACRO
       BAD    PARAM1,PARAM2,PARAM3=JJJ
 * PARAMETER 1:          PARAM1
 * PARAMETER 2:          PARAM2
 * PARAMETER 3:          PARAM3
BAD    ENDM                   ; End of macro definition.
       LIST   MAC             ; Listing expansion.
```

The call and expansion of the `BAD` macro is as follows:

```
       BAD    PARAM3=XKK      ; Macro call.
* PARAMETER 1:          PARAM3=KKK
* PARAMETER 2:
* PARAMETER 3:          JJJ
```

Macro `good` demonstrates the correct method for replacing parameters.

```
       macro
       good   param1,param2,param3=jjj
_* parameter 1:          param1
                               ; Null string.
_* parameter 2:          param2
                               ; Null string.
_* parameter 3:          param3
good   endm                    ; End of macro definition.
       list   mac              ; Listing expansion.
```

The call and expansion of the `good` macro is as follows:

```
       good   ,,param3=kkk     ; Macro call.
* parameter 1:                 ; Null string.
* parameter 2:                 ; Null string.
* parameter 3:          kkk
```

Macro `ALPHA` demonstrates the specification of an embedded parameter.

```
       MACRO                   ; EDIT=ON
       ALPHA  !PARAM           ; Appending a string.
_* FORMAL PARM:          PARAM
_* EMBEDDED PARM:        ABC_PARAM_DEFG
                               ; Concatenation off at call time.
ALPHA  ENDM                    ; End of macro definition.
       LIST   MAC              ; List expansion.
```

The call and expansion of the ALPHA macro is as follows:

```
      ALPHA  1                ; Macro call.
 * FORMAL PARM:          1
 * EMBEDDED PARM:        ABC1DEFG
```

CAL processes 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 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.

Operation definitions (OPDEF)
6.3

An operation definition (OPDEF) identifies a sequence of statements to be called later in the source program by an opdef call. Each time the opdef call occurs, the definition sequence is placed into 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.

You can replace any existing CAL machine instruction 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. 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:

register_mnemonic.*register_parameter*

- Expressions have been redefined for the opdef definition prototype statement, as follows:

@[*expression_parameter*]

These two exceptions allow you to specify parameters 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 that identifies the parameter in the opdef prototype statement must be identical to every character in the body of the opdef definition. This includes the case (uppercase, lowercase, or mixed case) of each character.

Because 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, which greatly increase the number of statements 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 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 as a macro and use fewer statements in the process.

The following example illustrates that an opdef can accept many different kinds of arguments from the opdef call:

```
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                 ; End of opdef definition.
```

The following example illustrates the calls and expansions of the previous example:

```
       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.
ONE    =      1                ; Define symbols.
TWO    =      2
THREE  =      3
       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 to 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`, and `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 if they take a valid form. The arguments passed to `REG1`, `REG2`, and `REG3` in the fourth call are `1`, `2`, and `THREE`, respectively.

The following example illustrates how to use an opdef to limit the number of statements required in a defined sequence:

```
      MACRO
      $IF      REG1,COND,REG2  ; Macro prototype statement.
      .
      .
      .
$IF   ENDM                     ; End of macro definition.
      .
      .
      .
      $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.

Because 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. You can specify the arguments for the registers in a number of ways and still be recognized as S register arguments by the opdef.

```
        opdef
example $if   s.reg1,eq,s.reg2; Opdef definition statement.
_* Register1: reg1
_* Register2: reg2
example endm                  ; End of opdef definition.
        list mac              ; Listing expansion.
```

The following are the calls and expansions of the preceding example:

```
        $if            s6,eq,s.3
* Register1:  6
* Register2:  3
```

If an opdef occurs within the global definitions part of a program segment, it is defined as global. 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. You can reference a global definition anywhere within an assembler program after it has been defined.

You can specify the `OPDEF` pseudo instruction 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`):

```
example  opdef
         s.reg         @exp  ; Prototype statement.
         a.reg         @exp  ; New machine instruction.
example  endm                ; End of opdef definition.
         list          mac   ; Listing expansion.
```

The following are the calls and expansions of the preceding example:

```
         s1            2     ; Opdef call.
         a.1           2     ; New machine instruction.
```

Opdef definition
6.3.1

The OPDEF pseudo instruction is the first statement of an opdef definition. Although an opdef is constructed much like a macro, an opdef is defined by an opdef statement, not by a functional name.

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 one 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. No special syntax forms exist 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 subfields within the operand field.

The format of the opdef definition is as follows:

```
name      OPDEF
[loc]     defsynres      defsynop
          LOCAL          [name][,[name]]
          .
          .
          .
name      ENDM
```

The variables in the opdef definition are described as follows:

- *name*

  name identifies the opdef definition and has no association with functionals that appear 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*

  loc specifies an optional location field parameter. *loc* must meet the requirements for names as described in subsection 4.2, page 67.

- *defsynres*

  defsynres specifies the definition syntax for the result field. It 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

  To specify an initial register on the opdef prototype statement, use one of the following four syntax forms for *initial-registers*:

[*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*]]

  The elements of an initial register definition are as follows:

  – *prefix*

    prefix is optional and can be either a right parenthesis (() or a right bracket ([).

– *register-prefix*

register-prefix is optional and case-sensitive. 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. It can be specified as any of the following characters:

```
<     >     #<    #>    #     #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
```

– *register*

register is required. It can be any simple or complex register. Simple registers are any of the following:

`CA`, `CE`, `CI`, `CL`, `MC`, `RT`, `SB`, `SM`, `VL`, `VM` or `XA`

The complex registers are designated in the opdef definition in the form: *register_designator.register_parameter*. The *register_designator* for complex registers can be any of the following:

`A`, `B`, `SB`, `SM`, `SR`, `ST`, `S`, `T` or `V`

The *register-parameter* is a 1- to 8-character identifier composed of identifier characters.

When you specify a *simple register* or a *complex register* mnemonic 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.

– *register-separator*

register-separator is optional and case-sensitive. It can be one of the following (when a register separator is specified on an opdef call, it is recognized by the opdef definition without regard to the case):

```
+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
```

– *register-expression-separator*

The optional *register-expression-separator* can be designated by any of the following:

`)`, `]`, `&`, `!`, `\`, `#<`, `#>`, `<`, `>`, `+`, `-`, `*` or `/`

– *special-register-separator*

The optional *special-register-separator* is specified as `#`.

– *register-ending*

The optional *register-ending* is specified using one of the following three syntax forms:

$register_1$ [*register-separator* [$register_2$ [*suffix*]]]
register [*register-expression-separator* [*register-or-expression* [*suffix*]]]
$register_1$ [*special-register-separator* [$register_2$ [*suffix*]]]

The $register_1$, *register-separator*, $register_2$, *suffix*, *register*, and *register-expression-separator* elements are described previously under *initial-register.*

The optional *register-or-expression* can be a register or an expression. If *register* is not specified, *expression* is required. If *expression* is not specified, *register* is required.

expression has been redefined for the opdef prototype statement, as *expression-parameter*. *expression-parameter* is an identifier that must begin with the at symbol (@). The @ can be followed by 0 to 7 identifier characters.

special-register-separator is specified as #.

– *expression-ending*

expression-ending is specified as follows:

expression [*expression_separator* [*register-or-expression* [*suffix*]]]

expression is required and has been redefined for the opdef prototype statement, as follows:

expression-parameter

expression-parameter is an identifier that must begin with the at symbol (@). The @ can be followed by 0 to 7 identifier characters.

expression-separator can be one of the following:

`)`, `]`, `&`, `!`, `\`, `=`, `#<`, or `#>`

The optional *register-or-expression* can be a register or an expression. If *register* is not specified, *expression* is required. If *expression* is not specified, *register* is required.

A mnemonic is a 1- to 8-character identifier that must begin with a letter (A through Z or a through z), a decimal digit (0 through 9), or one of the following characters: `$`, `%`, `&`, `'`, `*`, `+`, `-`, `.`, `/`, `:`, `=`, `?`, `\`, `` ` ``, `|`, or `~`. Optional characters 2 through 8 can be the at symbol (@) or any of the previously mentioned characters.

Initial-expression specifies an *initial-expression* on the opdef prototype statement, use one of the following syntax forms for *initial-expressions*:

[*prefixl* [*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*]]

The elements of the initial expression are described as follows:

– *prefix*

prefix is optional and can be either a right parenthesis (() or a right bracket ([).

– *expression-prefix*

expression-prefix is optional and can be any of the following:

`<`, `>`, `#<`, or `#>`

– *expression*

expression is required and has been redefined for the opdef prototype statement, as follows:

expression-parameter

expression-parameter is an identifier that must begin with the at symbol (`@`). The `@` can be followed from 0 to 7 identifier characters.

– *expression-separator*

expression-separator is optional and can be one of the following:

`)`, `]`, `&`, `!`, `\`, `<`, `>`, `#<`, or `#>`

– *register-ending* and *expression-ending*

register-ending and *expression-ending* are the same for initial expressions as for initial registers.

- *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 the definition of *defsynres,* earlier in this subsection.

Opdef calls
6.3.2

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 special syntax for registers and expressions that was required on the opdef definition does not extend to the opdef call.

The format of the opdef call is as follows:

locarg	*callsynres*	*callsynop*

The variables associated with the opdef call are described as follows:

- *locarg*

  locarg is an optional location field argument. It can consist of any characters and is terminated by a space (embedded spaces are illegal).

  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*

  callsynres specifies the result field syntax for the opdef call. It 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 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.

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*

  callsynop specifies the operand field syntax for the opdef call. It can consist of zero, one, two, or three subfields, and it must have the same syntax as specified in the operand field of the opdef definition prototype statement.

  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.

  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 in the form @*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 period (`.`) 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 4 octal digits.
  - If the *register-parameter* is of the form `.`*integer-constant* or `.`*symbol*, the actual argument is an *integer-constant* or a symbol.

The following opdef definition shows a scalar floating-point divide sequence:

```
fdv   opdef                         ; Scalar floating-point divide prototype
                                    ; statement.
L     s.r1     s.r2/fs.r3
      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
```

The following example illustrates the opdef call and expansion of the preceding example:

```
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
```

The following opdef definition, call, and expansion define a conditional jump where a jump occurs if the A register values are equal:

```
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                     ; End of opdef definition.
      LIST     MAC             ; Listing expansion.
```

The following example illustrates the opdef call and expansion of the preceding example (The expansion starts on line 2.):

```
      JEQ    A3,A6,GO          ; Opdef call.
      A0     A3-A5
*     JAZ    GO                ; Expression is expected.
```

The opdef in the following example illustrates how an opdef can redefine an existing machine instruction:

```
EXAMPLE  OPDEF
         S.REG           @EXP  ; Opdef protype instruction.
         A.REG           @EXP  ; New instruction.
EXAMPLE  ENDM                  ; End of opdef definition.
         LIST            MAC   ; Listing expansion.
```

The following example illustrates the opdef call and expansion of the preceding example:

```
        S1    2                ; Opdef call.
        A.1   2                ; New instruction.
```

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:

```
regchg opdef
lbl    s.reg1 s.reg2          ; Opdef prototype statement.
lbl    =      *               ; Left-shift if lbl is left off.
       s.reg2 s.reg1          ; Register s2 gets register s1.
regchg endm                   ; End of opdef definition.
       list   mac             ; Listing expansion.
```

The following example illustrates the opdef call and expansion of the preceding example:

```
       s1   s2                ; Opdef call.
   =     *                    ; Left-shift if lbl is left off.
       s.2  s.1               ; Register s2 gets register s1.
```

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.

The following example illustrates the case insensitivity of the register and register-prefix:

```
CASE   OPDEF
       S1     #Pa2            ; Prototype statement.
       .
       .
       .
CASE   ENDM
```

The following example illustrates the opdef calls of the preceding example:

```
       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.
```

Duplication (DUP)

6.4

The DUP pseudo instruction defines a sequence of code that is assembled repetitively immediately following the definition. The sequence of code is assembled the number of times specified on the DUP pseudo instruction. The sequence of code to be repeated consists of the statements following the DUP pseudo instruction and any optional LOCAL pseudo instructions. Comment statements are ignored. The sequence to be duplicated ends when the statement count is exhausted or when an ENDDUP pseudo instruction with a matching location field name is encountered.

The DUP pseudo instruction only accepts one type of formal parameter. That parameter must be specified with the LOCAL pseudo instruction.

You can specify the DUP pseudo instruction 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.

The format of the DUP pseudo instruction is as follows:

[*dupname*]	DUP	*expression*[, [*count*]]

The variables associated with the DUP pseudo instruction are described as follows:

- *dupname*

  dupname specifies an optional name for the dup sequence. It is required if the *count* field is null or missing. If no count field is present, *dupname* must match an ENDDUP name. The sequence field in the DUP pseudo instruction itself represents the nested dup level and appears in columns 89 and 90 on the listing. For a description of sequence field nest level numbering, see subsection 6.1, page 128.

  The *dupname* variable must meet the requirements for names as described in subsection 4.2, page 67.

- *expression*

  expression is an absolute expression with a positive value that specifies the number of times to repeat the code sequence. All symbols, if any, must be defined previously. If the current base is mixed, octal is used for the expression. If the value is 0, the code is skipped. You can use a STOPDUP to override the given expression.

  The *expression* operand must meet the requirements for expressions as described in subsection 4.7, page 94.

- *count*

  count is an optional absolute expression with positive value that specifies the number of statements to be duplicated. All symbols (if any) must be defined previously. 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, and dup or echo sequences.

The *count* operand must meet the requirements for expressions as described in subsection 4.7, page 94.

In the following example, the code sequence following the `DUP` pseudo instruction will be repeated 3 times. There are 5 statements in the sequence.

```
     DUP      3,5
     LOCAL    SYM1,SYM2       ; LOCAL pseudo instruction not counted.
*Asterisk comment; not counted
     S1       1               ; First statement is definition.
*Asterisk comment; not counted
     INCLUDE  ALPHA           ; INCLUDE pseudo instruction  not
                              ; counted.
```

The following is the file, `ALPHA`:

```
     S2     3                 ; Second statement in definition.
     S4     4                 ; Third statement in definition.
*Asterisk comment; not counted
     S5     5                 ; Fourth statement in definition.
     S6     6                 ; Fifth statement in definition.
```

In the following example, the two `con` pseudo instructions are duplicated three times immediately following the definition:

```
        list            dup
example dup             3     ; Definition.
        con             1
        con             2
example enddup
```

The following example illustrates the expansion of the preceding example:

```
        con    1
        con    2
        con    1
        con    2
        con    1
        con    2
```

Duplicate with varying argument (ECHO)
6.5

The ECHO pseudo instruction defines 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 after 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.

You can use the STOPDUP pseudo instruction to override the repetition count determined by the number of arguments in the longest argument list.

You can specify the ECHO pseudo instruction 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.

The format of the ECHO pseudo instruction is as follows:

dupname	ECHO	[*name*=*argument*][, [*name*=*argument*]]

The variables associated with the `ECHO` pseudo instruction are described as follows:

- *dupname*

  dupname specifies the required name of the echo sequence. It 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 subsection 4.2, page 67.

- *name*

  name specifies the formal parameter name. It must be unique. There can be none, one, or more formal parameters. *name* must meet the requirements for names as described in subsection 4.2, page 67.

- *argument*

  argument specifies a list of actual arguments. The list can be one argument or a parenthesized list of arguments.

  A single argument is any ASCII character up to but not including the element separator, a space, a tab (new format only), 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. A legal ASCII string is 4(5). An illegal ASCII string is (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 subsection 4.7, page 94.

In the following example, the ECHO pseudo instruction is expanded twice immediately following the definition:

```
         LIST              DUP
EXAMPLE  ECHO              PARAM1=(1,3),PARAM2=(2,4)
                                 ; Definition.
         CON               PARAM1
                                 ; Gets 1 and 3.
         CON               PARAM2
                                 ; Gets 2 and 4.
EXAMPLE  ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
       CON    1                  ; Gets 1 and 3.
       CON    2                  ; Gets 2 and 4.
       CON    3                  ; Gets 1 and 3.
       CON    3                  ; Gets 1 and 3.
       CON    4                  ; Gets 2 and 4.
```

In the following example, the echo pseudo instruction is expanded once immediately following the definition with two null arguments.

```
         list              dup
example  echo              param1=,param2=()
                                 ; ECHO with two null parameters.
_*Parameter 1 is:            'param1'
_*Parameter 2 is:            'param2'
example  enddup
```

The following illustrates the expansion of the preceding example:

```
*Parameter 1 is:      ''
*Parameter 2 is:      ''
```

Ending a macro or operation definition (ENDM)

6.6

An ENDM pseudo instruction terminates the body of a macro or opdef definition. If ENDM is used within a MACRO or OPDEF definition with a different name, it has no effect.

You can specify the ENDM pseudo instruction only 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.

The format of the ENDM pseudo instruction is as follows:

func	ENDM	ignored

The *func* variable associated with the ENDM pseudo instruction identifies the name of the macro or opdef definition sequence. It must be a valid identifier or the equal sign. *func* must match the functional that appears 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.

Premature exit from a macro expansion (EXITM)

6.7

The EXITM pseudo instruction 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 are also terminated immediately. If such an expansion does not exist, the EXITM pseudo instruction issues a caution level listing message and does nothing.

You can specify the EXITM pseudo instruction 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.

The format of the `EXITM` pseudo instruction is as follows:

ignored	EXITM	ignored

In the following example of a macro call, the macro expansion is terminated immediately by the `EXITM` pseudo instruction and the second comment is not included as part of the expansion:

```
       macro
       alpha
_*First comment
       exitm
_*Second comment
alpha  endm
       list   mac
```

The following example illustrates the expansion of the preceding example:

```
       alpha                   ; Macro call
*First comment
       exitm
```

Ending duplicated code (ENDDUP)
6.8

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. If `ENDDUP` is used within a `DUP` or `ECHO` definition with a different location field name, it has no effect. `ENDDUP` has no effect on a dup definition terminated by a statement count; in this case, `ENDDUP` is counted.

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.

The format of the `ENDDUP` pseudo instruction is as follows:

dupname	`ENDDUP`	ignored

The *dupname* variable associated with the `ENDDUP` pseudo instruction specifies the required name of a dup sequence. *dupname* must meet the requirements for names as described in subsection 4.2, page 67.

Premature exit of the current iteration of duplication expansion (NEXTDUP) 6.9

The `NEXTDUP` pseudo instruction stops the current iteration of a duplication sequence indicated by a `DUP` or an `ECHO` pseudo instruction. Assembly of the current repetition of the dup sequence is terminated immediately and the next repetition, if any, is begun.

Assembly of the current iteration of the innermost duplication expansion with a matching location field name is terminated immediately. 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. If a file also is being included at expansion time within the duplication expansion it is terminated immediately.

You can specify the `NEXTDUP` pseudo instruction 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.

The format of the `NEXTDUP` pseudo instruction is as follows:

[*dupname*]	`NEXTDUP`	ignored

The optional *dupname* variable specifies the 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 instruction 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 instruction does nothing.

Stopping duplication (STOPDUP) 6.10

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 no location field name exists, `STOPDUP` will terminate the innermost dup or echo sequence. `STOPDUP` does not affect the definition of the code sequence that will 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 that will be terminated, these expansions also are terminated immediately. If a file also is being included at expansion time within the duplication expansion that will be terminated, the inclusion of that file is terminated immediately.

You can specify the `STOPDUP` pseudo instruction 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.

The format of the `STOPDUP` pseudo instruction is as follows:

[*dupname*]	`STOPDUP`	ignored

The *dupname* variable associated with the `STOPDUP` pseudo instruction specifies the name of a dup sequence. If the name is present but does not match any existing duplication expansion, or, if the name is not present and a duplication expansion does not currently exist, a caution-level listing message is issued and the pseudo instruction does nothing. *dupname* must meet the requirements for names as described in subsection 4.2, page 67.

The following example uses a `DUP` pseudo instruction to define an array with values 0, 1, and 2:

```
S     =     W*
      DUP   3,1
      CON   W.*-S
```

The following illustrates the expansion of the preceding example:

```
      CON   W.*-S
      CON   W.*-S
      CON   W.*-S
```

In the following example the `ECHO` and `DUP` pseudo instructions define a nested duplication:

```
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
```

Note: The following expansion is not generated by CAL, but it is included to show the expansion of the previously nested duplication expansion.

The following example illustrates the expansion of the preceding example:

```
                                   ; In the first call of the echo, the A
                                   ; and B parameters are used.
      A.0      B.0                 ; DUPJK generates the A.O gets register
      .        .                   ; B.O 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 DUPJK.
      .        .
      A.1      B.0                 ; DUPJK generates register A.i gets
      .        .                   ; register B.0 through register A.i gets
      .        .                   ; register B.64 instructions.
      .        .
      A.7      B.64                ;  i is 1 to 7.
      S.0      T.O                 ; In the second expansion of the echo
      .        .                   ; pseudo instruction 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
```

In the following example the `STOPDUP` pseudo instruction terminates duplication:

```
      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
```

The following example illustrates the expansion of the preceding example:

```
T       SET     T+1
        CON     T
T       SET     T+1
        CON     T
T       SET     T+1
A       STOPDUP
```

In the following example a `STOPDUP` pseudo instruction is used to terminate a `DUP` immediately:

```
DNAME    DUP             3
_* First comment
         STOPDUP
_* Second comment
DNAME    ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
* First comment
         STOPDUP
```

The following example is similar to the previous example except `NEXTDUP` replaces `STOPDUP`. The current iteration is terminated immediately when the `NEXTDUP` pseudo instruction is encountered.

```
DNAME    DUP             3
_* First comment
         NEXTDUP
_* Second comment
DNAME    ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
* First comment
         NEXTDUP
* First comment
         NEXTDUP
* First comment
         NEXTDUP
```

Specifying local unique character string replacements (LOCAL)
6.11

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 known only in the macro, opdef, dup, or echo at expansion time. The most common usage of the `LOCAL` pseudo instruction is for defining symbols, but the `LOCAL` pseudo instruction is not restricted to the definition of symbols. Local pseudo instructions within a macro, opdef, dup, or echo header are not part of the macro definition.

On each macro or 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` and `ECHO` pseudo instructions, except for intervening comment statements.

You can specify the `LOCAL` pseudo instruction only 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.

The format of the `LOCAL` pseudo instruction is as follows:

ignored	`LOCAL`	[*name*][, [*name*]]

The *name* variable associated with the `LOCAL` pseudo instruction specifies formal parameters that must be unique and will be rendered local to the definition. *name* must meet the requirements for names as described in subsection 4.2, page 67.

The following example demonstrates that all formal parameters must be unique:

```
      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
```

The following example demonstrates how a unique character string is generated for each parameter defined by the `LOCAL` pseudo instruction:

```
       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                    ; End of macro definition.
       list   mac              ; Listing expansion.
```

The following example illustrates the call and expansion from the preceding example:

```
         string              ; Macro call.
 %%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`).

Synonymous operations (OPSYN) 6.12

The `OPSYN` pseudo instruction defines an operation that is synonymous with another macro or pseudo instruction operation. The functional name in the location field is defined as being the same as the functional name in the operand field. You can redefine any pseudo instruction or macro in this manner.

The functional name in the location field can be a currently defined macro or pseudo instruction 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 instruction occurs within the global part of an assembler segment, and it is local if the `OPSYN` pseudo instruction appears within an assembler module of a segment. You can reference global operations 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.

The format of the `OPSYN` pseudo instruction is as follows:

```
func1      OPSYN      [func2]
```

The *func1* variable associated with the `OPSYN` pseudo instruction specifies a required functional name. It must be a valid functional name. The name of a defined operation such as a pseudo instruction or macro, or the equal sign. *func*1 must not be blank and must meet the requirements for functional names.

The *func2* variable specifies an optional functional name. It must be the name of a defined operation or the equal sign. If *func*2 is blank, *func*1 becomes a do-nothing pseudo instruction.

In the following example, the macro definition includes the `OPSYN` pseudo instruction that redefines the `IDENT` pseudo instruction:

```
IDENTT   OPSYN           IDENT
         MLEVEL          ERROR ; Eliminates the warning error that is
                               ; issued because the IDENT pseudo
                               ; instruction 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
```

The following example illustrates the `OPSYN` call and expansion (The expansion starts on line 2.):

```
     IDENTT    A
     LIST      LIS,OFF,NXRF
A    LIST      LIS,ON,XRF      ; Processed if LIST=NAME on CAL control
                               ; statement.
     IDENTT    A
```

In the following example, the `first` macro illustrates that a functional can be redefined many times:

```
         macro
         first
         s1              1
         s2              2
         s3              s1+2
first    endm
second   opsyn           first
                            ; second is the same as first.
third    opsyn           second
                            ; third is the same as second.
```

The following example includes the Opdef calls and expansions from the preceding example:

```
     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
```

In the following example, the functional `EQU` is defined to perform the same operation as `=`:

```
EQU  OPSYN  =            ; EQU is defined to
                         ; perform the
                         ; operation that the
                         ; = pseudo
                         ; instruction
                         ; performs.
```

Pseudo Instruction Descriptions [A]

This appendix lists the pseudo instructions presented throughout section 5, page 117, in alphabetical order for easy reference. The pseudo instructions are listed at the left margin. The paragraphs to the right of each pseudo instruction name describe the pseudo instruction.

Note: You can specify pseudo instructions in uppercase or lowercase, but not in mixed case.

Throughout this appendix, pseudo instructions with ignored fields (location or operand) are defined as follows:

```
ignored    pseudox
```

ignored — The assembler ignores the location field of this statement. If the field is not empty and all of the characters in the field are skipped until a blank character is encountered, a caution-level message is issued. The first nonblank character following the blank character is assumed to be the beginning of the result field.

pseudox — Pseudo instruction with a blank location field.

```
           pseudoy      ignored
```

pseudoy — Pseudo instruction with a blank operand field.

ignored — The assembler ignores the location field of this statement. If the field is not empty and all of the characters in the field are skipped until a blank character is encountered, a caution-level message is issued. The first nonblank character following the blank character is assumed to be the beginning of the comment field.

=

The equate symbol (=) when used as a pseudo instruction defines a symbol with the value and attributes determined by the expression. The symbol is not redefinable.

You can specify the = pseudo instruction 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.

The format of the = pseudo instruction is as follows:

[*symbol*]	=	*expression*[, [*attribute*]]

The *symbol* variable represents 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 subsection 4.3, page 69.

All symbols found within *expression* must have been previously defined. The *expression* operand must meet the requirements for an expression as described in subsection 4.7, page 94.

The *attribute* variable specifies a parcel (`P`), word (`W`), or value (`V`) attribute. If present, it is used instead of the expression's attribute. If a parcel-address attribute is specified, an expression with word-address attribute is multiplied by four; if word-address attribute is specified, an expression with parcel-address attribute is divided by four. You cannot specify a relocatable expression as having value attribute.

In the following example, the symbol `SYMB` is assigned the value of `A*B+100/4`. The following illustrates the use of the = pseudo instruction:

```
SYMB    =           A*B+100/4
```

ALIGN

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 type of machine for which CAL is targeting code (see the `cpu=`*primary* option on the CAL invocation statement) determines the size of the offset.

Machine type	Octal offset (words/parcels)
CRAY C90	40/200
CRAY J90	40/200
CRAY T90	40/200
CRAY Y-MP	40/200

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. When an `ALIGN` pseudo instruction is encountered, the section relative to the current location counter is aligned.

If the location counter is currently positioned at an instruction buffer boundary, alignment is not performed. If the section that is being aligned has a type of `STACK` or `TASK COMMON` or has a location of local memory, a warning message is issued.

The `ALIGN` pseudo instruction is restricted to sections that have a type of instruction, data, or both. 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.

The format of the `ALIGN` pseudo instruction is as follows:

```
[symbol]     ALIGN     ignored
```

The *symbol* variable is optional. 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 subsection 4.3, page 69.

The octal value in the output listing immediately to the left of the location field indicates the number of full parcels skipped.

The following example illustrates the use of the ALIGN pseudo instruction:

```
L     =     *
      J     A
A     ALIGN
```

BASE

The BASE pseudo instruction specifies the base of numeric data as octal, decimal, or mixed when the base is not explicitly specified by an O′, D′, or X′ prefix. The default is decimal.

You can specify the BASE pseudo instruction anywhere in a program segment. However, if the BASE pseudo instruction is located within a definition or skipping section, it is not recognized as a pseudo instruction.

The format of the BASE pseudo instruction is as follows:

ignored	BASE	*option*/*

The *option* variable specifies the numeric base of numeric data. It is a required single character specified as follows:

- O or o (Octal)
- D or d (Decimal)
- M or m (Mixed)

Numeric data is assumed to be octal, except for numeric data used for the following (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 subsection 4.4.2.3, page 84.

When the asterisk (*) is used with the BASE pseudo instruction, the numeric base reverts to the base 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 instructions specified are released, a caution-level message is issued, and the default mode (decimal) is used.

The following example illustrates the use of the BASE pseudo instruction:

```
       BASE     0           ; 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.
       .
       .
       .
       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 (default)
```

BITP

The BITP pseudo instruction sets the bit position to the value specified relative to bit 0 of the current parcel. 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.

The format of the BITP pseudo instruction is as follows:

```
ignored     BITP     [expression]
```

The *expression* variable is optional. 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 through 16 (decimal). All symbols within *expression* (if any) must be defined previously. If the current base is mixed, decimal is used.

The *expression* operand must meet the requirements for an expression as described in subsection 4.7, page 94.

The value generated in the code field of the listing is equal to the value of the expression.

The following example illustrates the use of the BITW pseudo instruction:

```
        vwd    d'16/0     ; Fill first 16 bits with 0.
        vwd    6/o'12     ; Fill next 6 bits with 001100.
        bitp   0          ; Reset the pointer to bit 0 of parcel B.
        vwd    6/o'12     ; 001100 from previous word is ORed with
                          ; 001010
```

In the preceding example, `O'14` and `O'12` are `OR`ed and the result is `1110`:

Figure 28 through Figure 31 illustrate what happens when CAL assembles the previous example. ↑ represents the current bit position, and ^ indicates an uninitialized bit.

When CAL encounters the `VWD d'16/0` instruction, the following is stored in parcel A:

Figure 28. `BITP` example – zoning parcel A

The following is stored in parcel b when `VWD 6/0'14` is assembled:

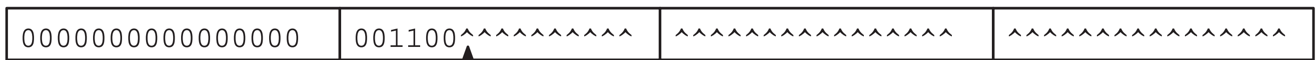


Figure 29. `BITP` example – parcel b set by `VWD` instruction

The pointer is reset to bit `0` of parcel B when the `bitp 0` instruction is encountered, as follows:

Figure 30. `BITP` example – resetting the pointer

The next instruction, `VWD 6/O'12`, causes `001010` (`o'12`) to be `OR`ed with the first 6 bits of parcel B (`001100`), producing `001110`, which is stored, as follows:

Figure 31. `BITP` example – result of a `BITP` followed by a `VWD`

BITW

The BITW pseudo instruction resets the current bit position to the value specified, relative to bit 0 of the current word. If the current bit position is not bit 0, 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 bit 0, the BITW pseudo instruction with a value of 64 does not force a word boundary, and the instruction following BITW is assembled at bit 0 of the current word.

If the origin and location counters are set lower than the current value, any code previously generated in the overlapping part 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.

The format of the BITW pseudo instruction is as follows:

```
ignored      BITW       [expression]
```

The *expression* variable is optional. 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 defined previously. If the current base is mixed, decimal is used.

The *expression* operand must meet the requirements for expressions as described in subsection 4.7, page 94.

The value generated in the code field of the listing is equal to the value of the expression.

The following example illustrates the use of the BITW pseudo instruction:

```
         BITW     D'39
```

BLOCK

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 instruction defines a mixed local section in which both code and/or data can be stored. The section is assigned to central or common memory. For more information, see the description of the SECTION pseudo instruction on page 263 of this appendix.

You must specify the BLOCK pseudo instruction 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.

The format of the BLOCK pseudo instruction is as follows:

```
          BLOCK        [lname]/*
```

The *lname* variable is optional and identifies the block. It indicates which section is used for assembling code until the occurrence of the next BLOCK or COMMON pseudo instruction.

This long name is restricted in length depending on the type of loader table that is currently generating the assembler. If the name is too long, the assembler issues an error message.

The *lname* operand must meet the requirements for long names as described subsection 4.3.1, page 70.

The asterisk (*) indicates that 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 were released when a BLOCK * is encountered, CAL issues a caution-level message and uses the main section.

The following example illustrates the use of the BLOCK pseudo instruction:

```
        .                  ; Main section is in use.
        .
        .
        BLOCK   A          ; Use section A
        .
        .
        .
        BLOCK              ; Use main section
        .
        .
        .
        BLOCK   *          : Return to use of section A.
```

BSS

The BSS pseudo instruction reserves a block of memory in a section. A forced word boundary occurs and the number of words specified by the operand field expression is reserved. This pseudo instruction does not generate data. To reserve the block of memory, the location and origin counters are increased.

You must specify the BSS pseudo instruction 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.

The format of the BSS pseudo instruction is as follows:

[*symbol*]	BSS	[*expression*]

The *symbol* variable is optional. It is assigned the word address of the location counter after the force word boundary occurs. *symbol* must meet the requirement for symbols as described in subsection 4.3, page 69.

The *expression* variable is 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.

The *expression* operand must meet the requirements for an expression as described in subsection 4.7, page 94.

The left margin of the listing shows the octal word count.

The following example illustrates the use of the BSS pseudo instruction:

```
        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.
```

BSSZ

The BSSZ pseudo instruction generates a block of words that contain 0's. When BSSZ is specified, a forced word boundary occurs, and the number of zeroed 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.

The format of the BSSZ pseudo instruction is as follows:

[*symbol*]	BSSZ	[*expression*]

The *symbol* variable represents an optional symbol. It is 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 subsection 4.3, page 69.

The *expression* variable represents 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 0's that will be generated. A blank operand field results in no data generation. The *expression* operand must meet the requirement for an expression as described in subsection 4.7, page 94.

The octal word count of a BSSZ is shown in the left margin of the listing.

CMICRO

The CMICRO pseudo instruction assigns a name to a character string. After 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.

The format of the CMICRO pseudo instruction is as follows:

name	CMICRO	[*string*[,[*exp*][,[*exp*][,[*case*]]]]]

The *name* variable is required and is assigned to the character string in the operand field. It has nonredefinable attributes. If *name* was 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 subsection 4.2, page 67.

The *string* variable represents an optional character string that 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 * can be specified as `'*'` or `****`).

The *exp* variable represents optional expressions. The first expression must be an absolute expression that indicates 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 expressions must meet the requirements for expressions as described in subsection 4.7, page 94.

The micro character string is terminated by the value of the first expression or the final apostrophe of the character string, whichever occurs first. If the first expression has a 0 or negative value, the string is considered empty. 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 defined previously. 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 optional *case* variable denotes the way 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*. You can specify *case* in uppercase, lowercase, or mixed case, and it must be one of the following:

- `MIXED` or `mixed`

  string is interpreted as you entered it and no case conversion occurs. This is the default.

- `UPPER` or `upper`

  All lowercase alphabetic characters in *string* are converted to their uppercase equivalents.

- `LOWER` or `lower`

  All uppercase alphabetic characters in *string* are converted to their lowercase equivalents.

COMMENT

The COMMENT pseudo instruction defines a character string of up to 256 characters that will 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 most recent one is used. If the last comment differs from the previous comment, a caution-level message is issued.

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

You must specify the COMMENT pseudo instruction 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.

The format of the COMMENT pseudo instruction is as follows:

ignored	COMMENT	[*del-char*[*string-of-ASCII*]*del-char*]

The *del-char* variable designates the delimiter character. It 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 that a single such character will be included in the character string.

The *string-of ASCII* variable is an optional ASCII character string of any length.

The following example illustrates the use of the COMMENT pseudo instruction:

```
IDENT    CAL
COMMENT  'COPYRIGHT CRAY RESEARCH, INC.  1992'
COMMENT  -CRAY Y--MP computer system-
COMMENT  @ABCDEF@@FEDCBA@
END
```

COMMON

The COMMON pseudo instruction establishes a common section or resumes a previous section. Each section has its own location, origin, and bit position counters.

This pseudo instruction defines a common section that can be referenced by another program module. Instructions are not allowed. The section is assigned to common memory. For more information, see subsection 5.4, page 120.

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, ZEROCOM, or TASKCOM. If duplicate location field names are specified, an error-level message is issued.

For a description of unnamed (blank) COMMON, see section 3, page 33.

You must specify the COMMON pseudo instruction 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.

The format of the COMMON pseudo instruction is as follows:

```
ignored     COMMON     [lname]/*
```

The *lname* variable specifies the optional long name of the common section to be defined. *lname* must meet the requirements for long names as described in subsection 4.3.1, page 70.

The long name is restricted in length depending on the type of loader table the assembler is currently generating. If the name is too long, the assembler issues an error message.

Unlabeled common sections have specific restrictions. For a detailed description of blank COMMON sections, see section 3, page 33.

The asterisk (*) specifies that 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 were released when a COMMON * is encountered, CAL issues a caution-level message and uses the main section.

The following example illustrates the use of the BLOCK pseudo instruction:

```
      .                    ; Main section ins use.
      .
      .
      COMMON   FIRST       ; Labeled common section FIRST.
      .
      .
      .
      COMMON               ; Blank common.
      .
      .
      .
      COMMON   *           ; Return to labeled common section FIRST.
      .
      .
      .
      COMMON   *           ; Return to the main section.
```

CON

The CON pseudo instruction generates one or more full words of binary data. This pseudo instruction always causes a forced 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.

The format of the CON pseudo instruction is as follows:

```
[symbol]        CON        [expression]{,[expression]}
```

The *symbol* variable is an optional symbol. It is 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 subsection 4.3, page 69.

The *expression* variable is an expression whose value will be inserted into one 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 subsection 4.7, page 94.

The following example illustrates the use of the CON pseudo instruction:

```
A     CON    O'7777017
      CON    A              ; Generates the
                            ; address of A.
```

DATA

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 forced 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 forced word boundary does not occur.

The DATA pseudo instruction is restricted to sections that have a type of data, constants, 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.

The format of the `DATA` pseudo instruction is as follows:

```
[symbol]        DATA        [data_item][,[data_item]]
```

The symbol variable represents an optional symbol that is 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 subsection 4.3, page 69.

The *data_item* variable represents numeric or character data. *data_item* must meet the requirements for a data item as described in subsection 4.4, page 76.

The `DATA` pseudo instruction works with the actual number of bits given in the data item.

In the following example, unlabeled data items are stored in the next available bit position (see Figure 32):

```
       IDENT    EXDAT
       DATA     'abcd'*       ; Unlabeled data item 1.
       DATA     'efgh'        ; Unlabeled data item 2.
       END
```

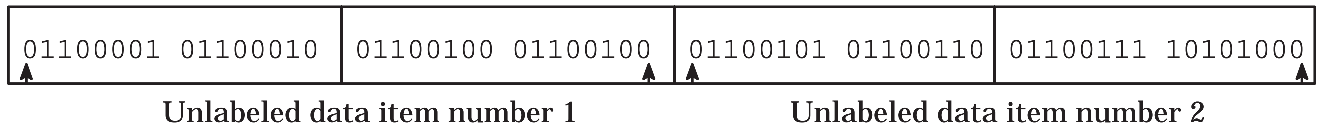


Figure 32. Storage of unlabeled data items

In the following example, labeled data items cause a forced word boundary (see Figure 33, page 208):

```
        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.
```

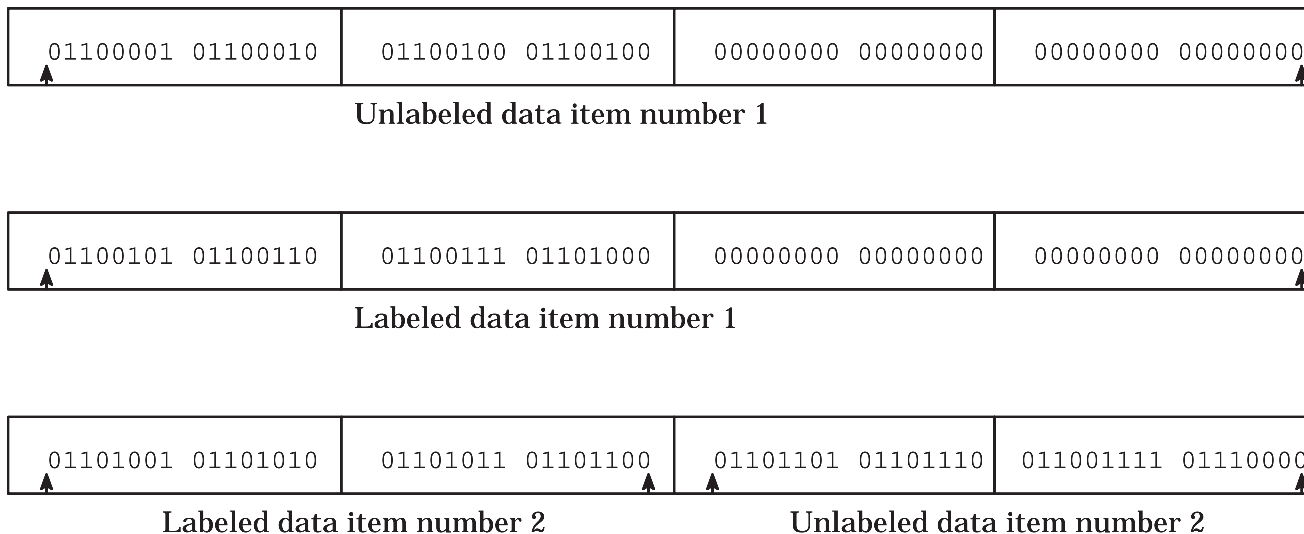


Figure 33. Storage of labeled and unlabeled data items

In the following example, if no forced word boundary occurs, data is stored bit by bit in consecutive words (see Figure 34). The following *data-item* is defined with the CDC character set (6 bits per character).

```
       IDENT  EXDAT
       DATA   C'ABCDEFGHIJK'* ; Unlabeled data item 1.
```

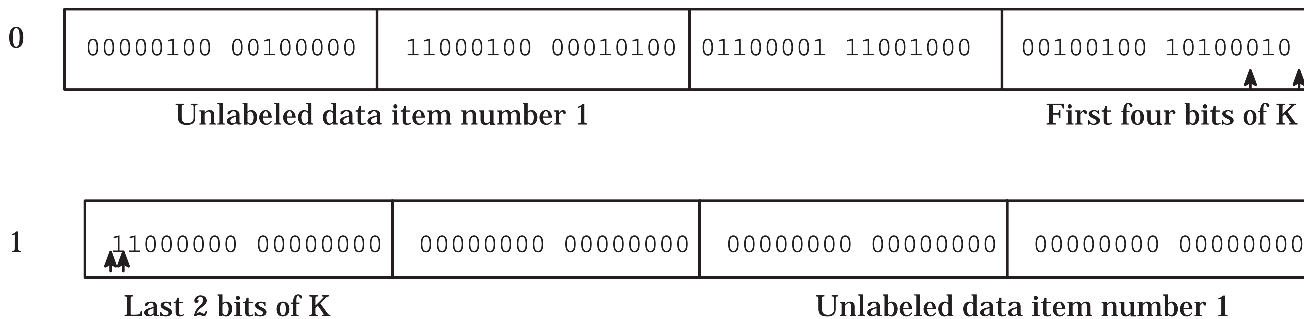


Figure 34. Storage of CDC character data item

The following example shows the code generated by each source statement:

```
      IDENT   EXAMPLE
      DATA    0'5252,A'ABC'R    ; 0000000000000000005252
                                ; 0000000000000020241103
      DATA    'ABCD'            ; 0405022064204010020040
      DATA    'EFGH'            ; 0425062164404010020040
      DATA    'ABCD'*           ;  040502206420
      DATA    'EFGH'*           ;  10521443510
      DATA    'ABCD'12R         ; 0000000000000000000000
                                ;  040502206420
      DATA    'EFGHIJ'*         ;            10521443510
                                ;  044512
LL2   DATA    'ABCD'            ; 0405022064204010020040
      DATA    100               ; 0000000000000000000144
      DATA    1.25E-9           ; 0377435274616704302142

      DATA    'THIS IS A MESSAGE'*L
                                ; 0521102225144022251440
                                ; O40440232425232464O5O7
                                ;  0424
      VWD     8/0               ;  000
      END
```

DBSM

The `DBSM` pseudo instruction generates a named label entry in the debug symbol tables with the type specified.

The format of the `DBSM` pseudo instruction is as follows:

[ignored]	`DBSM`	*TYPE*=`symbol`

TYPE is specified as either `ATP` or `BOE` (after the prologue or beginning of epilogue). *symbol* is user defined and marks these two points in the code. The *symbol* can appear anywhere in the code, but the address that is entered into the debug symbol table is the address of where the pseudo instruction appears in the code. This pseudo instruction is ignored unless you specify the debug option on the command line.

The following example illustrates the use of the DBSM pseudo instruction:

```
     IDENT    TEST
     ENTRY    FRED
FRED =        *
     BSSZ     16         ; Fake prolog.
     S4       S4
CHK  =        *
     DBSM     ATP=FRED   ; Should be the same as CHK address.
     A1       S1
     A1       S1
     A1       S1
     DBSM     BOE=FRED   ; Address should be the same as the next
                         ; instruction
     S1       5
     J        B00
```

From the debugger, you can do a stop in FRED to generate a breakpoint at CHK. A call to this routine from a program executing in the debugger stops the execution.

DECMIC

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.

You can specify DECMIC 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.

You can specify the `DECMIC` pseudo instruction 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.

The format of the `DECMIC` pseudo instruction is as follows:

name	`DECMIC`	[*expression*$_1$][,[*expression*$_2$[,[*option*]]]]

name is assigned to the character string that represents the decimal value of *expression*$_1$ and has redefinable attributes. *name* must meet the requirements for identifiers as described in subsection 4.2, page 67.

expression1 is optional and represents the 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 0 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 0 is used when creating the micro string. The *expression*$_1$ operand must meet the requirements for expressions as described in subsection 4.7, page 94.

expression2 is optional and provides a positive character count less than or equal to decimal 20. If this parameter is present, the necessary leading zeros or blanks (depending on *option)* are supplied to provide the requested number of characters. If specified, *expression*$_2$ 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 decimal is used. *expression*$_2$ must meet the requirements for expressions as described in subsection 4.7, page 94.

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.

option represents the type of fill characters (`ZERO` for zeros or `BLANK` for spaces) to be used if the second expression is present and fill is needed. The default is `ZERO`. You can enter *option* in mixed case.

The following example illustrates the use of the `DECMIC` and `MICSIZE` pseudo instructions:

```
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.†
```

† Generated by CAL

The following example demonstrates the `ZERO` and `BLANK` options with positive and negative strings:

```
       BASE   D                 ; The base is decimal
ONE    DECMIC 1,2
_*     "ONE"                    ; Returns 1 in 2 digits.
*      01                       ; Returns 1 in 2 digits.
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.
```

DMSG

The `DMSG` pseudo instruction issues a comment level diagnostic message that contains the string found in the operand field, if a string exists. If the string consists of 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. For more information, see section 2, page 11.

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).

You can specify the DMSG pseudo instruction 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.

The format of the DMSG pseudo instruction is as follows:

ignored	DMSG	[*del-char*[*string-of-ASCII*]*del-char*]

The *del-char* variable represents the delimiting character. It must be a single matching character on both ends of the ASCII character string. Apostrophes and spaces are not legal delimiters; all other ASCII characters are allowed. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

The *string-of ASCII* variable represents the ASCII character string that will be printed to the diagnostic file. A maximum of 80 characters is allowed.

> **Note:** Using the DMSG pseudo instruction for assembly timings can be deceiving. For example, if the DMSG pseudo instruction is inserted near the beginning of an assembler segment, more time could elapse (from the time that CAL begins assembling the segment to the time the message is issued) than you might have expected.

DUP

The DUP pseudo instruction introduces a sequence of code that is assembled repetitively a specified number of times The duplicated code immediately follows the DUP pseudo instruction.

The DUP pseudo instruction is described in detail in subsection 6.4, page 171.

ECHO

The ECHO pseudo instruction defines a sequence of code that is assembled zero or more times immediately following the definition.

The ECHO pseudo instruction is described in detail in subsection 6.5, page 174.

EDIT

The EDIT pseudo instruction toggles the editing function on and off within a program segment. Appending (^ in the new format) and continuation (, in the old format) are not affected by the EDIT pseudo instruction. The current editing status is reset at the beginning of each segment to the editing option specified on the CAL invocation statement. For a description of statement editing, see subsection 3.3, page 41.

You can specify the EDIT pseudo instruction 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.

The format of the EDIT pseudo instruction is as follows:

ignored	EDIT	*/*option*

The *option* variable turns editing on and off. *option* can be specified in uppercase, lowercase, or mixed case, and it can be one of the following:

- ON (enable editing)
- OFF (disable editing)
- No entry (reverts to the format specified on the CAL invocation statement)

An asterisk (*) 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 * initiates 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 were released, a caution-level message is issued and the default is used.

EJECT

The `EJECT` pseudo instruction causes the beginning of a new page in the output listing. `EJECT` is a list control pseudo instruction and by default, is not listed. To include the `EJECT` pseudo instruction on the listing, specify the `LIS` option on the `LIST` pseudo instruction.

You can specify the `EJECT` pseudo instruction 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.

The format of the `EJECT` pseudo instruction is as follows:

ignored	`EJECT`	ignored

ELSE

The `ELSE` pseudo instruction terminates skipping initiated by the `IFA`, `IFC`, `IFE`, `ELSE`, or `SKIP` pseudo instructions with the same location field name. If statements are currently being skipped under control of a statement count, `ELSE` has no effect.

You can specify the `ELSE` pseudo instruction 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.

The format of the `ELSE` pseudo instruction is as follows:

name	`ELSE`	ignored

The *name* variable specifies a required name for a conditional sequence of code. *name* must meet the requirements for identifiers as described in subsection 4.2, page 67.

The following example illustrates the use of the `ELSE` pseudo instruction:

```
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 is SYM is
                                ; absolute.
BTEST    ELSE
         A1               SYM   ; Assemble if SYM is 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     ; Statement is included.
         S6               6     ; Statement is included.
Z        ENDIF
```

END

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

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

The format of the END pseudo instruction is as follows:

ignored	END	ignored

If the END pseudo instruction is found within a definition, a skip sequence, or an expansion, a message is issued indicating that the pseudo instruction 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

You can specify the END pseudo instruction only from within a program module. If the END pseudo instruction is valid and terminates a program module, it causes the assembler to take the following actions:

- Generates a cross-reference for symbols if the cross-reference list option is enabled and the listing is enabled
- Clears and resets the format option
- Clears and resets the edit option
- Clears and resets the message level
- Clears and resets all list control options
- Clears and resets the default numeric base
- Discards all qualified, redefinable, nonglobal, and %% symbols
- Discards all qualifiers
- Discards all redefinable and nonglobal micros

- Discards all local macros, opdefs, and local pseudos instructions (defined with an `OPSYN` pseudo instruction)
- Discards all sections

ENDDUP

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.

The `ENDDUP` pseudo instruction is described in detail in subsection 6.8, page 178.

ENDIF

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` does not affect skipping, which is controlled by a statement count.

You can specify the `ENDIF` pseudo instruction anywhere within a program segment. Skipping is terminated by an `ENDIF` pseudo instruction with a matching location field name. If the `ENDIF` pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction.

The format of the `ENDIF` pseudo instruction is as follows:

name	`ENDIF`	ignored

The *name* variable specifies a required name for a conditional sequence of code. *name* must meet the requirements for identifiers as described in subsection 4.2, page 67.

> **Note:** If an `END` pseudo instruction is encountered in a skipping sequence, an error message is issued and skipping is continued. You should not use the `END` pseudo instruction within a skipping sequence.

ENDM

An `ENDM` pseudo instruction terminates the body of a macro or opdef definition.

The `ENDM` pseudo instruction is described in detail in subsection 6.6, page 177.

ENDTEXT

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 instruction 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.

You can specify the ENDTEXT pseudo instruction 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.

The format of the ENDTEXT pseudo instruction is as follows:

```
ignored     ENDTEXT     ignored
```

The following example illustrates the use of the ENDTEXT pseudo instruction (with the TXT option off).

The following represents the source listing:

```
         IDENT            TEXT
A        =                2
TXTNAME  TEXT             'An example.'
B        =                3
C        =                4
         ENDTEXT
         A1               A
         A2               B
         END
```

The following represents the output listing:

```
         IDENT            TEXT
A        =                2
TXTNAME  TEXT             'An example.'
         A1               A
         A2               B
         END
```

ENTRY

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 or data or both. If the ENTRY pseudo instruction is found within a definition or skipping sequence, it is defined and not recognized as a pseudo instruction.

The format of the ENTRY pseudo instruction is as follows:

ignored	ENTRY	[*symbol*] , [*symbol*]

The *symbol* variable specifies the 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.

The length of the symbol is restricted depending on the type of loader table that the assembler is currently generating. If the symbol is too long, the assembler will issue an error message.

The *symbol* operand must meet the requirements for symbols as described in subsection 4.3, page 69.

The following example illustrates the use of the ENTRY pseudo instruction:

```
        ENTRY   EPTNME,TREG
        .
        .
        .
EPTNME =        *
TREG   =        O'17
```

ERRIF

The ERRIF pseudo instruction conditionally issues a listing message. If the condition is satisfied (true), the appropriate user-defined message is issued. If the level is not specified, the ERRIF pseudo instruction 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 appropriate user-defined message is issued.

You can specify the ERRIF pseudo instruction 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.

The format of the ERRIF pseudo instruction is as follows:

[*option*]	ERRIF	[*expression*] , *condition* , [*expression*]

The *option* variable used in the ERRIF pseudo instruction is the same as in the ERROR pseudo instruction. See the ERROR pseudo instruction for information.

Zero, one, or two expressions to be compared by *condition.* If one or both of the expressions are missing, a value of absolute 0 is substituted for every expression that is not specified. Symbols found in either of the expressions can be defined later in a segment.

The *expression* operand must meet the requirements for expressions as described in subsection 4.7, page 94.

The *condition* variable 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. You can enter *condition* in uppercase, lowercase, or mixed case, and it can be one of the following:

- `LT` (less than)

  The value of the first expression must be less than the value of the second expression.

- `LE` (less than or equal)

  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)

  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. Both expressions must 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.

- NE (not equal)

  The first expression must not equal the second expression. Both expressions cannot be absolute, or external relative to the same external symbol, or relocatable in the program section or the same common section. The word-address, parcel-address, or value attributes are not the same.

The ERRIF pseudo instruction does not compare the address and relative attributes. A CAUTION level message is issued.

The following example illustrates the use of the ERRIF pseudo instruction:

```
P       ERRIF  ABC,LT,DEF
```

ERROR

The ERROR pseudo instruction unconditionally issues a listing message. If the level is not specified, the ERROR pseudo instruction issues an error level message. If the condition is not satisfied (FALSE), no message is issued.

You can specify the ERROR pseudo instruction 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.

The format of the ERROR pseudo instruction is as follows:

[*option*]	ERROR	ignored

The *option* variable specifies the error level. It can be entered in upper, lower, or mixed case. The following error levels are mapped directly into a user-defined message of the corresponding level:

COMMENT, NOTE, CAUTION, WARNING, or ERROR

The following levels are mapped into an error-level message:

C, D, E, F, I, L, N, O, P, R, S, T, U, V, or X

The following levels are mapped into warning-level messages:

`W`, `W1`, `W2`, `W3`, `W4`, `W5`, `W6`, `W7`, `W8`, `W9`, `Y1`, or `Y2`

Messages `C` through `Y2` provide compatibility with Cray Assembly Language, version 1 (CAL1).

CAL can produce five similar messages with differing levels (error, warning, caution, note, or comment). The `ERROR` pseudo instruction can be used to check for valid input and to assign an appropriate message.

In the following example, a user-defined error level message is specified:

```
ERROR    ERROR            ; ***ERROR*** Input is not valid
```

EXITM

The `EXITM` pseudo instruction immediately terminates the innermost nested macro or opdef expansion, if any, caused by either a macro or an opdef call.

The `EXITM` pseudo instruction is described in detail in subsection 6.7, page 177.

EXT

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 in the `EXT` instruction are defined as unqualified symbols that have relative attributes of external and specified address.

You can specify the `EXT` pseudo instruction anywhere within a program module. If the `EXT` pseudo instruction is found within a definition or skipping sequence, it is defined and not recognized as a pseudo instruction.

The format of the `EXT` pseudo instruction is as follows:

```
ignored      EXT       [symbol:[attribute]],[symbol:[attribute]]
```

The variables associated with the `EXT` pseudo instruction are described as follows:

- *symbol*

  The *symbol* variable specifies the name of zero, one, or more external symbols. Each of the names must be an unqualified symbol that has a relative attribute of external and the corresponding address attribute.

  The length of the symbol is restricted depending on the type of loader table that the assembler is currently generating. If the symbol is too long, the assembler will issue an error message.

  The *symbol* operand must meet the requirements for symbols as described in subsection 4.3, page 69.

- *attribute*

  The *attribute* variable specifies either the attribute *address-attribute* or *linkage-attribute* as follows:

  – The *address-attribute* type is the address attribute that will be assigned to the external symbol; it can be one of the following:

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

– The *linkage-attribute* type is the linkage attribute that will be assigned to the external symbol. Linkage attributes can be specified in uppercase, lowercase, or mixed case, and they can be one of the following:

  `HARD (default)`

  `SOFT`

  If the *linkage-attribute* is not specified on the `EXT` pseudo instruction, 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 only when at least one other module has referenced that same external name as a hard reference.

  You conditionally reference a soft external name at execution time. If a soft external name was not 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: Typically, a soft external is used for references to large software packages (such as graphics packages) that may not be required in a particular load. When such code is required, load time is shorter and the absolute module is smaller in size. For most uses, however, hard externals are recommended.

The following example illustrates the use of the EXT pseudo instruction:

```
       IDENT   A
       .
       .
       .
       ENTRY   VALUE
VALUE  =       2.0
       .
       .
       .
       END
       IDENT   B
       EXT     VALUE
       CON     VALUE       ; The 64-bit external. External value 2.0 is
                           ; stored here by the loader.
       END
```

FORMAT

CAL supports both the CAL, version 1 (CAL1) statement format and a new statement format. The FORMAT pseudo instruction lets you 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, page 33.

You can specify the FORMAT pseudo instruction 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.

The format of the FORMAT pseudo instruction is as follows:

ignored	FORMAT	*/*option*

The *option* variable specifies old or new format. *option* can be specified in uppercase, lowercase, or mixed case, and it can be one of the following:

- `OLD` (old format)
- `NEW` (new format)
- No entry (reverts to the `EDIT` option specified on the CAL invocation statement)

An asterisk (`*`) 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 *` initiates 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 were released, a caution-level message is issued and the default is used.

IDENT

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

You must specify the `IDENT` pseudo instruction 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.

The format of the `IDENT` pseudo instruction is as follows:

ignored	`IDENT`	*lname*

The *lname* variable is the long name of the program module. *lname* must meet the requirements for long names as described in subsection 4.2, page 67.

The length of the long name is restricted depending on the type of loader table the assembler is currently generating. If the name is too long, the assembler issues an error message.

The following example illustrates the use of the `IDENT` psuedo instruction:

```
      IDENT    EXAMPLE    ; Beginning of the EXAMPLE program module
      .
      .                   ; Other code goes here
      .
      END                 ; End of the EXAMPLE program module
```

IFA

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.

You can specify the `IFA` pseudo instruction 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.

The format of the `IFA` pseudo instruction is as follows:

[*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*]]

The *name* variable specifies an optional name of a conditional sequence of code. A conditional sequence of code that is controlled by a name is ended by an `ENDIF` pseudo instruction with a matching name. To reverse the condition of a conditional sequence of code controlled by a name, use an `ELSE` pseudo instruction with a matching name. If both *name* and *count* are present, name takes precedence. *name* must meet the requirements for names as described in subsection 4.2, page 67.

The pound sign (#) is optional and negates the condition. If errors occur in the attribute condition, the condition is evaluated as if it were true. Although `#` does not change the condition, it does specify the if not condition.

The *exp-attribute* variable is a mnemonic that signifies an attribute of *expression*. *expression* must meet the requirement for an expression as described in subsection 4.7, page 94.

An expression has only one address attribute (`VAL`, `PA`, or `WA`) and relative attribute (`ABS`, `IMM`, `REL`, or `EXT`). An attribute also can be any of the following mnemonics preceded by a complement sign (`#`), indicating that the second subfield does not satisfy the corresponding condition. You can specify all of the following mnemonics in mixed case:

Mnemonic	Attribute
`VAL`	Value; requires all symbols within the expression to be defined previously.
`PA`	Parcel address; requires all symbols, if any, within the expression to be defined previously.
`WA`	Word address; requires all symbols, if any, within the expression to be defined previously.
`ABS`	Absolute; requires all symbols, if any, within the expression to be defined previously.
`IMM`	Immobile; requires all symbols, if any, within the expression to be defined previously.
`REL`	Relocatable; requires all symbols, if any, within the expression to be defined previously.
`EXT`	External; requires all symbols, if any, within the expression to be defined previously.
`CODE`	Immobile or relocatable; relative to a code section. `CODE` requires all symbols, if any, within the expression to be defined previously.
`DATA`	Immobile or relocatable; relative to a data section. `DATA` requires all symbols, if any, within the expression to be defined previously.

Mnemonic	Attribute
`ZERODATA`	Immobile or relocatable; relative to a zero data section. `ZERODATA` requires all symbols, if any, within the expression to be defined previously.
`CONST`	Immobile or relocatable; relative to a constant section. `CONST` requires all symbols, if any, within the expression to be defined previously.
`MIXED`	Immobile or relocatable; relative to a common section. `MIXED` requires all symbols, if any, within the expression to be defined previously.
`COM`	Immobile or relocatable; relative to a common section. `COM` requires all symbols, if any, within the expression to be defined previously.
`COMMON`	Immobile or relocatable; relative to a common section. `COMMON` requires all symbols, if any, within the expression to be defined previously.
`TASKCOM`	Immobile or relocatable; relative to a task common section. `TASKCOM` requires all symbols, if any, within the expression to be defined previously.
`ZEROCOM`	Immobile or relocatable; relative to a zero common section. `ZEROCOM` requires all symbols, if any, within the expression to be defined previously.
`DYNAMIC`	Immobile or relocatable; relative to a dynamic section. `DYNAMIC` requires all symbols, if any, within the expression to be defined previously.
`STACK`	Immobile or relocatable; relative to a stack section. `STACK` requires all symbols, if any, within the expression to be defined previously.
`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 defined previously.

Mnemonic	Attribute
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 defined previously. If EM is specified, 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 defined previously. If LM is specified for a Cray Research system, the condition always fails.
DEF	True if all symbols in the expression were defined previously; otherwise, the condition is false.

The *redef-attribute* variable specifies a redefinable attribute. The condition is true if the symbol following *redef-attribute* is redefinable; otherwise, the condition is false. Redefinable attribute is defined as follows:

Mnemonic	Attribute
SET	The *symbol* in the second subfield is a redefinable symbol. *symbol* must meet the requirements for a symbol as described in subsection 4.3, page 69.

The *reg-attribute* variable specifies a register attribute. *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 (,). If you specify REG, 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 in register designator.

The *micro-attribute* variable specifies an attribute of the micro specified by *mname*. *mname* must meet the requirements for identifiers as described in subsection 4.2, page 67. If you specify MIC, 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.
MICRO	The name in the second subfield is a micro name and the corresponding micro can be redefined.
CMICRO	The name in the second subfield is a micro name and the corresponding micro is constant.

The *count* variable specifies the statement count. It 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 used only 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.

The following example illustrates the use of the IFA pseudo instruction:

```
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.
```

IFC

The IFC pseudo instruction tests a pair of character strings for a condition under which code will 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 during evaluation of the string condition, the resulting condition is handled as if true and an appropriate listing message is issued.

You can specify the IFC pseudo instruction 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.

The format of the IFC pseudo instruction is as follows:

```
[name]     IFC     [string],condition,[string][,[count]]
```

The *name* variable specifies an optional name of a conditional sequence of code. A conditional sequence of code that is controlled by a name is ended by an ENDIF pseudo instruction with a matching name. To reverse the condition of a conditional sequence of code controlled by a name, use an ELSE pseudo instruction with a matching name. If both *name* and *count* are present, *name* takes precedence. *name* must meet the requirements for names as described in subsection 4.2, page 67.

The *string* variable specifies the character string that will 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 value of 0 is substituted for missing characters in the shorter string. Micros and formal parameters can be contained in the character strings.

The *string* operand is an optional ASCII character string that must be specified with one 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 will be included in the character string.

The following example compares the character strings `O'100` and `ABCD*`:

```
AIF IFC =O'100=,EQ,*ABCD***
```

The condition variable specifies the relation that will be satisfied by the two strings. You can enter *condition* in mixed case, and it must be one of the following:

- `LT` (less than)

  The value of the first string must be less than the value of the second string.

- `LE` (less than or equal)

  The value of the first string must be less than or equal to the value of the second string.

- `GT` (greater than)

  The value of the first string must be greater than the value of the second string.

- `GE` (greater than or equal)

  The value of the first string must be greater than or equal to the value of the second string.

- `EQ` (equal)

  The value of the 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.

The *count* variable specifies the statement count. It 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. The *count* operand is used

only 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.

The following examples illustrates the use of the `IFC` pseudo instruction. The first string is delimited by the at sign (`@`), and the second string is delimited by the percent sign (`%`). The first string is equal to the second string.

```
     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.
     END
```

In the next example, the first string is not equal to the second string, the two statements following the `IFC` are skipped.

```
     IDENT  TEST
EX1  IFC    @ABBCD@,EQ,@ABCD@ ; The condition is false.
                                ; Skipping occurs.
     S1     1                   ; Statement is skipped.
     S2     2                   ; Statement is skipped.
EX1  ENDIF                      ; End of skip sequence
     S3     3                   ; This statement is included regardless
                                ; of whether the condition is true or
                                ; false.
     END
```

IFE

The IFE pseudo instruction tests a pair of expressions for a condition. If the relation *(condition)* specified by the operation is satisfied, code is assembled. If *condition* is true, assembly resumes with the next statement; if *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 during the evaluation of 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.

You can specify the IFE pseudo instruction 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.

The format of the IFE pseudo instruction is as follows:

[*name*]	IFE	[*expression*] , *condition* , [*expression*] [, [*count*]]

The *name* variable specifies an optional name of a conditional sequence of code. A conditional sequence of code that is controlled by a name is ended by an ENDIF pseudo instruction with a matching name. To reverse the condition of a conditional sequence of code controlled by a name, use an ELSE pseudo instruction with a matching name. If both *name* and *count* are present, *name* takes precedence. *name* must meet the requirements for names as described in subsection 4.2, page 67.

The *expression* variables specify the expressions to be compared. All symbols in the expression must be defined previously. If an expression is not specified, the absolute value of 0 is used. *expressions* must meet the requirements for expressions as described in subsection 4.7, page 94.

The condition variable specifies the relation to be satisfied by the two strings. You can enter *condition* in mixed case, and it must be one of the following:

- `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)

  The value of the first expression must be less than or equal to the value of the second expression. The attributes are not checked.

- `GT` (greater than)

  The value of the first expression must be greater than the value of the second expression. The attributes are not checked.

- `GE` (greater than or equal)

  The value of the first expression must be greater than or equal to the value of the second expression. The attributes are not checked.

- `EQ` (equal)

  The value of the first expression must be equal to the value of the second expression. Both expressions must be one of the following:

  - Attributes must be the same
  - Immobile relative to the same section
  - Relocatable relative to the same section
  - External relative to the same external symbol.
  - The word-address, parcel-address, or value

- `NE` (not equal)

  The first expression and the second expression do not satisfy the conditions required for `EQ` described above.

The *count* variable specifies the statement count. It must be an absolute expression with a 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 used only 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.

The following example illustrates the use of the `IFE` pseudo instruction:

```
       IDENT  TEST
SYM1   =      0
SYM2   =      *
SYM3   SET    1000
SYM4   SET    500
NOTEQ  IFE    SYM1,EQ,SYM2    ; Condition fails, values are the same,
                              ; but the attributes are different.
       S1     SYM1            ; The ELSE sequence is assembled.
       S2     SYM2
NOTEQ  ELSE
       S1     SYM3            ; Statement is included.
       S2     SYM4            ; Statement is included.
NOTEQ  ENDIF                  ; End of conditional sequence.
       END
```

IFM

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 during the evaluation of the string condition, the resulting condition is handled as if true and an appropriate listing message is issued.

You can specify the `IFM` pseudo instruction 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.

The format of the `IFM` pseudo instruction is as follows:

```
[name]   IFM   [#]logical-name[,[count]]
[name]   IFM   numeric-name,condition,[expression] [,[count]]
```

The *name* variable specifies an optional name of a conditional sequence of code. A conditional sequence of code that is controlled by a name is ended by an `ENDIF` pseudo instruction with a matching name. To reverse the condition of a conditional sequence of code controlled by a name, use an `ELSE` pseudo instruction with a matching name. If both *name* and *count* are present, *name* takes precedence. *name* must meet the requirements for names as described in subsection 4.2, page 67.

The *logical-name* variable specifies the mnemonic that signifies a logical condition of the machine for which CAL is currently targeting code. If the logical name is preceded by a pound sign (`#`), its resultant condition is complemented. For a detailed list of the mnemonics, see the logical traits of the CPU option for the appropriate operating system in section 2, page 11.

The *numeric-name* variable specifies the mnemonic that signifies a numeric condition of the machine for which CAL is currently targeting code. For a detailed list of the mnemonics, see the numeric traits of the CPU option for the appropriate operating system in section 2, page 11. You can specify these mnemonics in mixed case.

The *condition* variable specifies the relation to be satisfied between the numeric name and the expression, if any. You can enter *condition* in mixed case, and it must be one of the following:

- `LT` (less than)

  The value of the numeric name must be less than the value of the expression.

- `LE` (less than or equal)

  The value of the numeric name must be less than or equal to the value of the expression.

- `GT` (greater than)

  The value of the numeric name must be greater than the value of the expression.

- GE (greater than or equal)

  The value of the numeric name must be greater than or equal to the value of the expression.

- EQ (equal)

  The value of the numeric name must be equal to the value of the expression.

- NE (not equal)

  The value of the numeric name must not equal the value of the expression.

The *expression* variable specifies the expression to be compared to the numeric name. All symbols in the expression must be defined previously 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. *expression* must meet the requirements for expressions as described in subsection 4.7, page 94.

The *count* variable specifies the statement count. It must be an absolute expression with a 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 used only 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.

The following example illustrates the use of the `IFM` pseudo instruction:

```
        ident  test
ex1     ifm    vpop            ; Assuming the condition is true,
        .                      ; skipping does occur within the IFM
        .                      ; part.
        .
ex1     ifm    numcpus,eq,4    ; Assuming the condition is false,
        .                      ; skipping occurs.
        .
        .
ex2     else                   ; Toggles the condition so that the else
        .                      ; part is not skipped.
        .
        .
ex2     endif
        end
```

INCLUDE

The `INCLUDE` psuedo instruction inserts a file at the current source position. The `INCLUDE` pseudo instruction always prepares the file for reading by opening it and positioning the pointer at the beginning.

You can use this pseudo instruction to include the same file more than once within a particular file.

You can also nest `INCLUDE` instructions. Because you cannot use `INCLUDE` recursively, you should review nested `INCLUDE` instructions for recursive calls to a file that you have already opened.

You can specify the `INCLUDE` pseudo instruction anywhere within a program segment. If the `INCLUDE` pseudo instruction occurs within a definition, it is recognized as a pseudo instruction and the specified 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 specified file is included in the skipping sequence. The `INCLUDE` pseudo instruction statement itself is not inserted into a defined sequence of code.

Note: The `INCLUDE` pseudo instruction can be forced into a definition or skipped sequence of code. When editing is enabled, `INCLUDE` is expanded during execution and the file is read in at that point. This method is not recommended because formal parameters are not substituted correctly into statements when the `INCLUDE` macro is expanded during execution.

If using this method, insert an underscore (_) anywhere within the pseudo instruction, as follows: `IN_CLUDE`.

If editing is disabled during execution, `INCLUDE` is not expanded.

The format of the `INCLUDE` pseudo instruction is as follows:

ignored	`INCLUDE`	*filename*

The *filename* variable is 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 cannot open the file, a listing message is issued.

filename must be specified with one matching character on each end. Any ASCII character other than a comma or space can be used. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

In the following examples, the module named `INCTEST` contains an `INCLUDE` pseudo instruction. The file to be included is named `DOG` and the `CAT` file is included within the `DOG` file.

The `INCTEST` module is as follows:

```
       IDENT    INCTEST
       INCLUDE  *DOG*          ; Call file DOG with INCLUDE.
       END
```

The file `DOG` contains the following:

```
S1       1                 ; Register S1 gets 1.
INCLUDE  'CAT'             ; Call file CAT with INCLUDE.
S2       2                 ; Register S2 gets 2.
```

The file `CAT` contains the following:

```
S3       3                 ; Register S3 gets 3.
```

The expansion of the `INCTEST` module is as follows:

```
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
```

The following example demonstrates that it is illegal to include a file recursively within nested `INCLUDE` instructions.

The `INCTEST` module is as follows:

```
IDENT    INCTEST
INCLUDE  *DOG*             ; Call file DOG with INCLUDE.
END
```

The file `DOG` contains the following:

```
S1       1                 ; Register S1 gets 1.
INCLUDE  'CAT'             ; Call file CAT with INCLUDE.
S2       2                 ; Register S2 gets 2.
```

The file CAT includes the following:

```
      S3       3                ; Register S3 gets 3.
      INCLUDE  -DOG-            ; Illegal. If file B was included by
                                ; file A, it cannot include file A.
```

The following example demonstrates that it is legal to include a file more than once if it is not currently being included.

The INCTEST module is as follows:

```
      IDENT    INCTEST
      INCLUDE  *DOG*            ; Call file DOG with INCLUDE.
      INCLUDE  *DOG*            ; Call file DOG with INCLUDE.
      END
```

The file DOG contains the following:

```
      S1       1                ; Register S1 gets 1.
      S2       2                ; Register S2 gets 2.
```

The expansion of the INCTEST module is as follows:

```
      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
```

LIST

The LIST pseudo instruction controls the listing. LIST is a list control pseudo instruction and by default, is not listed. To include the LIST pseudo instruction on the listing, specify the LIS option on this instruction. An END pseudo instruction resets options to the default values.

You can specify the LIST pseudo instruction 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.

The format of the LIST pseudo instruction is as follows:

[*name*]	LIST	[*option*]{,[*option*]}/*

The name variable specifies the optional list name. *name* must meet the requirements for identifiers as described in subsection 4.2, page 67.

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 instructions with a matching name are not ignored. LIST pseudos instructions with a blank location field are always processed.

The *option* variable specifies that a particular listing feature be enabled or disabled. All option names 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 instruction. If you use the no-list option on the CAL invocation statement, all LIST pseudo instructions in the program are ignored.

There can be zero, one, or more options specified or an *. If no options are specified, OFF is assumed. The allowed options are described as follows:

- ON (enables source statement listing)

  Source statements and code generated are listed (default).

- OFF (disables source statement listing)

  While this option is selected, only statements with errors are listed. If the LIS option is enabled, listing control pseudo instructions are also listed.

- ED (enables listing of edited statements)

  Edited statements are included in the listing file (default).

- NED (disables listing of edited statements)

  Edited statements are not included in the listing file.

- XRF (enables cross-reference)

  Symbol references are accumulated and a cross-reference listing is produced (default).

- NXRF (disables cross-reference)

  Symbol references are not accumulated. If this option is selected when the END pseudo instruction is encountered, no cross-reference is produced.

- XNS (includes nonreferenced local symbols in the reference)

  Local symbols that were not referenced in the listing output are included in the cross-reference listing (default).

- NXNS (excludes nonreferenced local symbols from the cross-reference)

  If this option is selected when the END pseudo instruction is encountered, local symbols that were not referenced in the listing output are not included in the cross-reference.

- LIS (enables listing of the listing pseudo instructions)

  The LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, and ENDTEXT psuedo instructions are included in the listing.

- NLIS (disables listing of the listing pseudo instructions)

  The LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, and ENDTEXT psuedo instructions are not included in the listing (default).

- TXT (enables global text source listing)

  Each statement following a TEXT pseudo instruction is listed through the ENDTEXT instruction if the listing is otherwise enabled.

- NTXT (disables global text source listing)

  Statements that follow a TEXT pseudo instruction through the following ENDTEXT instruction are not listed (default).

- MAC (enables 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 (disables listing of macro and opdef expansions)

  Statements generated by macro and opdef calls are not listed (default).

- MBO (enables 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 (disables 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 (default).

  Note: Source statements containing a micro reference (see MIC and NMIC options) or a concatenation character are listed before editing regardless of whether this option is enabled or disabled.

- MIC (enables listing of generated statements before editing)

  Statements that are generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, and that 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` (disables 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 (default).

  Note: Conditional statements (see `NIF` and `NMIF` options) and skipped statements in source code are listed regardless of whether this option is enabled or disabled.

- `MIF` (enables 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` (disables 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 (default).

- `DUP` (enables 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` (disables listing of duplicated statements)

  Statements generated by `DUP` and `ECHO` are not listed (default).

The asterisk (`*`) reactivates the `LIST` pseudo instruction in effect before the current `LIST` pseudo instruction was specified within the current program segment. Each occurrence of a `LIST` pseudo instruction other than `LIST` initiates a new listing control. Each `LIST` releases the current listing control and reactivates the listing control that preceded the current list control. If all specified listing controls were released when a `LIST *` is encountered, CAL issues a caution-level message and uses the defaults for listing control.

LOC

The LOC pseudo instruction sets 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 or data, or both. 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.

The format of the LOC pseudo instruction is as follows:

ignored	LOC	[*expression*]

The *expression* variable is optional and represents the new value of the location counter. If the expression does not exist, the counter is reset to the absolute value of 0. If the expression does exist, all symbols (if any) must be defined previously. If the current base is mixed, octal is used as the base.

The *expression* operand 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. The *expression* operand must meet the requirements for an expression as described in subsection 4.7, page 94.

The following example illustrates the use of the LOC pseudo instruction:

```
        ORG     Q.*+1000
        LOC     200
LBL     A1      0
        .
        .
        .
        J       LBL
```

Note: In the preceding example, the code is generated and loaded at location `W.*+10000` and the user must move it to absolute location `200` before execution.

LOCAL

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 known only in the macro, opdef, dup, or echo at expansion time. The most common usage of the `LOCAL` pseudo instruction is for defining symbols, but it is not restricted to the definition of symbols.

The `LOCAL` pseudo instruction is described in detail in subsection 6.11, page 184.

MACRO

The `MACRO` pseudo instruction marks the beginning of a sequence of source program instructions saved by the assembler for inclusion in a program when called for by the macro name.

Macros are described in detail in subsection 6.2, page 134.

MICRO

The `MICRO` pseudo instruction assigns a name to a character string. The assigned name can be redefined. You can reference and redefine a redefinable micro after its initial 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 segments that follow.

You can specify the `MICRO` pseudo instruction 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.

The format of the `MICRO` pseudo instruction is as follows:

name	`MICRO`	[*string*[,[*exp*][,[*exp*][,[*case*]]]]]

The *name* variable is required and is assigned to the character string in the operand field. It has redefinable attributes. If *name* was previously defined, the previous micro definition is lost. *name* must meet the requirements for identifiers as described in subsection 4.2, page 67.

The *string* variable represents an optional character string that 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 `*` can be specified as `'*'` or `****`).

The *exp* variable represents optional expressions. The first expression must be an absolute expression that indicates 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 expressions must meet the requirements for expressions as described in subsection 4.7, page 94.

The micro character string is terminated by the value of the first expression or the final apostrophe of the character string, whichever occurs first. If the first expression has a 0 or negative value, the string is considered empty. 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 that indicates the micro string's starting character. All symbols, if any, must be defined previously. 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 optional *case* variable denotes the way 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*. You can specify *case* in uppercase, lowercase, or mixed case, and it must be one of the following:

- `MIXED` or `mixed`

  string is interpreted as entered and no case conversion occurs. This is the default.

- `UPPER` or `upper`

  All lowercase alphabetic characters in *string* are converted to their uppercase equivalents.

- `LOWER` or `lower`

  All uppercase alphabetic characters in *string* are converted to their lowercase equivalents.

The following example illustrates the use of the `MICRO` pseudo instruction:

```
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'
```

MICSIZE

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.

† CAL has edited these lines

You can specify the `MICSIZE` pseudo instruction 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.

The format of the `MICSIZE` pseudo instruction is as follows:

[*symbol*]	`MICSIZE`	*name*

The *symbol* variable specifies an 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 subsection 4.3, page 69.

The name variable represents the name of a micro string that has been previously defined. *name* must meet the requirements for identifiers as described in subsection 4.2, page 67.

MLEVEL

The `MLEVEL` pseudo instruction changes the level of messages received in your source listing. If the `ML` option on the CAL invocation statement differs from the option on the `MLEVEL` pseudo instruction, the invocation statement overrides the pseudo instruction.

If the option accompanying the `MLEVEL` pseudo instruction is not valid, a diagnostic message is generated and `MLEVEL` is set to the default value.

You can specify the `MLEVEL` pseudo instruction 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.

The format of the MLEVEL pseudo instruction is as follows:

ignored	MLEVEL	[*option*]/*

The *option* variable specifies an optional message level. It can be entered in uppercase, lowercase, or mixed case, it must be one of the following levels (the default is WARNING):

- ERROR (enables error-level messages only)
- WARNING (enables warning- and error-level messages)
- 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)

The asterisk (*) reactivates the message level in effect before the current message level was specified within the current program segment. Each occurrence of an MLEVEL pseudo instruction other than MLEVEL * initiates a new message level. Each MLEVEL * releases the current message level and reactivates the message level that preceded the current message level. If all specified message levels have been released when an MLEVEL * is encountered, CAL issues a caution-level message to alert you to the situation and then reverts to the default level, warning.

NEXTDUP

The NEXTDUP pseudo instruction stops the current iteration of a duplication sequence indicated by a DUP or an ECHO pseudo instruction. Assembly of the current repetition of the dup sequence is terminated immediately and the next repetition, if any, is begun.

The NEXTDUP pseudo instruction is described in detail in subsection 6.9, page 179.

OCTMIC

The `OCTMIC` pseudo instruction converts the value of an expression to a character string that is assigned a redefinable micro name. The character string that the pseudo instruction generates is represented as an octal number. The final length of the micro string is inserted into the code field of the listing.

You can specify `OCTMIC` 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.

You can specify the `OCTMIC` pseudo instruction 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.

The format of the `OCTMIC` pseudo instruction is as follows:

name	`OCTMIC`	[*expression*$_1$] [`","`[*expression*$_2$[`","`[*option*]]]]

The *name* variable is required and specifies the name of the micro. *name* must meet the requirements for identifiers as described in subsection 4.2, page 67.

The *expression*$_1$ variable is an optional expression and is equal to *name*. If specified, *expression*$_1$ must have an address attribute of value and a relative attribute of absolute. All symbols used must be 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 0 is used in creating the micro string. The *expression*$_1$ operand must meet the requirements for expressions as described in subsection 4.7, page 94.

expression$_2$ provides a positive character count less than or equal to decimal 22. If this parameter is present, leading 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 attribute 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 octal value of the first expression. The *expression*$_2$ operand must meet the requirements for expressions as described in subsection 4.7, page 94.

option represents the type of fill characters (ZERO for zeros or BLANK for spaces) that will be used if the second expression is present and fill is needed. The default is ZERO. You can enter *option* in mixed case.

The following example illustrates the use of the OCTMIC pseudo instruction:

```
      IDENT    EXOCT
      BASE     0                 ; The base is octal.
ONE   OCTMIC   1,2
_*    “ONE”                      ; Returns 1 in 2 digits.
*     01                         ; Returns 1 in 2 digits.
TWO   OCTMIC   5*7+60+700,3
_*    “TWO”                      ; Returns 1023 in 3 digits.
*
*     023                        ; Returns 1023 in 3 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
```

OPDEF

The `OPDEF` pseudo instruction marks the beginning of an operation definition (`opdef`). The opdef identifies a sequence of statements to be called later in the source program by an opdef call. Each time the opdef call occurs, the definition sequence is placed into the source program.

The `OPDEF` pseudo instruction is described in detail in subsection 6.3, page 154.

OPSYN

The `OPSYN` pseudo instruction defines an operation that is synonymous with another macro or pseudo instruction operation.

The `OPSYN` pseudo instruction is described in detail in subsection 6.12, page 186.

ORG

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 forced word boundaries occur before the counter is reset. `ORG` can change the current working section without modifying the section stack.

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

The format of the `ORG` pseudo instruction is as follows:

ignored	`ORG`	[*expression*]

The *expression* variable is 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 defined previously. If the current base is mixed, octal is used as the base.

The expression cannot have an address attribute of parcel, a relative attribute of absolute or external, or a negative value. The *expression* operand must meet the requirements for an expression as described in subsection 4.7, page 94.

The following example illustrates the use of the ORG pseudo instruction:

```
        ORG     W.*+0'200
```

QUAL

A QUAL pseudo instruction begins or ends a code sequence in which all symbols defined are qualified by a qualifier specified by the QUAL pseudo instruction 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. The QUAL pseudo instruction must not occur before an IDENT pseudo instruction.

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

You must specify the QUAL pseudo instruction from within a program module. If the QUAL pseudo instruction is found within a definition or skipping sequence, it is defined and is not recognized as a pseudo instruction.

At the end of each program segment, all qualified symbols are discarded.

The format of the QUAL pseudo instruction is as follows:

```
ignored     QUAL     */[name]
```

The *name* variable is optional and indicates whether symbols will be qualified or unqualified and, if qualified, indicates the qualifier to be used. The *name* operand must meet the requirements for names as described subsection 4.3.1, page 70.

The *name* operand 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 tries to find it in the currently active qualifier. If the qualified symbol is not defined within the current qualifier, CAL tries to find it in the list of unqualified symbols. If both of these searches fail, the symbol is undefined.

An unqualified symbol can be referenced explicitly 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 asterisk (*) 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 activates the most recent prior qualification. If the QUAL * statement is encountered and all specified qualifiers are released, a caution-level message is issued and succeeding symbols are defined as being unqualified.

The following example illustrates the use of the `QUAL` pseudo instruction:

```
* 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 if ABC is defined within qualifier
                            ; QNAME1
C       IFA  DEF,/QNAME2/ABC; Test if /QNAME2/ABC is defined within
                            ; qualifier QNAME2.
        .
        .
        .
```

SECTION

The SECTION pseudo instruction establishes or resumes a section of code. The section can be common or local, depending on the options found in the operand field. Each section has its own location, origin, and bit-position counters.

You must specify the SECTION pseudo instruction 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.

The format of the SECTION pseudo instruction is as follows:

```
[lname]   SECTION   [type][","[location]][","[ENTRY]]
[lname]   SECTION   [location][","[type]][","[ENTRY]]
[lname]   SECTION   [type][","[ENTRY]][","[location]]
[lname]   SECTION   [location][","[ENTRY]][","[type]]
[lname]   SECTION   [ENTRY][","[location]][","[type]]
[lname]   SECTION   [ENTRY][","[type]][","[location]]
ignored   SECTION   *
```

The variables associated with the SECTION pseudo instruction are described as follows:

- *lname*

  The *lname* variable is optional and names the section. *lname* must meet the requirements for long names as described in subsection 4.3.1, page 70.

  The length of the long name is restricted depending on the type of loader table that the assembler is currently generating. If the name is too long, the assembler issues an error message.

- *type*

  The type variable specifies the type of section. It can be specified in uppercase, lowercase, or mixed case. *type* can be one of the following (for a description of local sections, see subsection 3.6.1, page 54):

  – MIXED

    Defines a section that permits both instructions and data. MIXED is the default type for the main section initiated by the IDENT pseudo instruction. If *type* is not specified, MIXED is the default. The loader treats a MIXED section as a local section.

  – CODE

    Restricts a section to instructions only; data is not permitted. The loader treats a CODE section as a local section.

  – DATA

    Restricts a section to data only (CON, DATA, BSSZ, and so on); instructions are not permitted. The loader treats the DATA section as a local section.

  – ZERODATA

    Neither instructions nor data are allowed within this section. The loader treats a ZERODATA section as a local section. At load time, all space within a ZERODATA section is set to 0.

  – CONST

    Restricts a section to constants only (CON, DATA, BSSZ, and so on); instructions are not permitted. The loader treats the CONST section as a local section.

– 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. Because 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 that CAL produces.

Note: Accessing data from a stack section is not as straightforward as accessing data directly from memory. For more information about stacks, see the *UNICOS Macros and Opdefs Reference Manual*, publication SR–2403.

– 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, ZEROCOM, or TASKCOM. If duplicate location field names are specified, an error level message is issued.

For a description of unnamed (blank) COMMON, see subsection 3.6.2, page 55.

– 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, ZEROCOM, or TASKCOM. If duplicate location field names are specified, an error-level message is issued.

For a description of blank DYNAMIC, see subsection 3.6.2, page 55.

– ZEROCOM

Defines a common section that can be referenced by another program module. Neither instructions nor data are permitted within a ZEROCOM section; only storage reservation can be defined.

At load time, all uninitialized space within a ZEROCOM section is set to 0. If a COMMON section with the same name contains the initialized text that was referenced by another module that will be loaded, portions of a ZEROCOM section can be explicitly initialized to values other than 0.

ZEROCOM must always be named. The location field that names a ZEROCOM section cannot match the location field name of a previously defined section with a type of COMMON, DYNAMIC, ZEROCOM, or TASKCOM. If duplicate location field names are specified, an error level message is issued.

– 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 that are local to a task. Data also can 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 instruction that defines the section. This symbol is not redefinable, has a value of 0, an address attribute of word, and a relative attribute that is

relocatable relative to the section. The loader relocates this symbol, and it is used as an offset into an execution time task common table. The word at which it points within this table contains the address of the base of the task common section in memory.

All symbols 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, but relocation does not occur after assembly. These symbols can be used as offsets 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 that CAL produces. For a description of local sections, see subsection 3.6.1, page 54.

`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`, `ZEROCOM`, or `TASKCOM`. If duplicate location field names are specified, an error level message is issued.

Note: Accessing data from a task common section is not as straightforward as accessing data directly from memory. For more information about task common, see the *CF90 Fortran Language Reference Manual*, publication SR–3902.

- *location*

  The kind of memory to which the section is assigned can be uppercase, lowercase, or mixed case, and it must be:

  `CM` Central or common memory (default).

- `ENTRY`

  Sets a bit in the Program Descriptor table to direct `segldr` to create an entry point at the same address as the first word of the section.

- *

  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 were released when a `SECTION *` is encountered, CAL issues a caution-level message and uses the main section.

  When *type* and/or *location* are not specified, `MIXED` and common memory are used by default.

If *type* and/or *location* are not specified, the defaults are `MIXED` for *type* and `CM` for *location.* Because a module within a program segment is initialized without a name, these defaults, when acting together, force this initial section entry to become the current working section.

If the section name and attributes are previously defined, the `SECTION` pseudo instruction makes the previously defined section entry the current working section. If the section name and attributes are not defined, the `SECTION` pseudo instruction tries to create a new section with the name and attributes. The following restrictions apply when a new section is created:

- A section of the type `TASKCOM`, `COMMON`, `ZEROCOM`, and a section with a specified entry must always have a location field name.
- If a section with a type of `COMMON`, `DYNAMIC`, `ZEROCOM`, 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`, `ZEROCOM`, or `TASKCOM`.

The following example illustrates the use of the `SECTION` pseudo instruction:

```
       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
       S1      1               ; the Main section.
       .
       .
       .
dsect  sectiondata             ; This section is defined with a name of
       .                       ; dsect, a type of data, and location of
       .                       ; common memory.
       .
       con     3               ; Data is permitted in dsect.
       bszz    2               ; Data is permitted in dsect.
       .
       .
       .
       S2      S3              ; CAL generates an error-level message
       .                       ; because instructions are illegal in a
       .                       ; section with a type of data.
       .
csect  sectioncommon           ; 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.
       S2      A1              ; CAL generates an error-level message,
       .                       ; 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.
```

(continued)

```
       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.
       .
       .
 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 that
                            ; are specified with the same name
       .                    ; therefore, are, considered unique if they
       .                    ; are specified with different types.
       .
       s1     s2            ; Instructions are permitted in dsect.
 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.
       .
       section*             ; The current section reverts to the main
       .                    ; section
       .
       .
       con    2             ; CAL generates an error-level message
       .                    ; because data is not permitted in a
       .                    ; section with a type of code.
       .
       section*             ; This current section reverts to the main
       .                    ; section.
       .
       .
```

(continued)

```
 csect  sectiondynamic      ; CAL generates an error-level message,
                            ; because the loader does not treat
                            ; sections with types of common, dynamic,
                            ; and taskcom as local sections Specifying
                            ; a section with a previously defined name
                            ; is illegal when the accompanying type
                            ; does not define a local section.
        end
```

SET

The SET pseudo instruction resembles the = pseudo instruction; however, a symbol defined by SET is redefinable.

You can specify the SET pseudo instruction 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.

The format of the SET pseudo instruction is as follows:

[*symbol*]	SET	*expression*[, [*attribute*]]

The *symbol* variable specifies an optional unqualified symbol. The symbol is implicitly qualified by the current qualifier. A symbol defined with the SET pseudo instruction can be redefined with another SET pseudo instruction, but the symbol must not be defined prior to the first SET pseudo instruction. The location field can be blank. *symbol* must meet the requirements for symbols as described in subsection 4.3, page 69.

All symbols found within *expression* must have been previously defined. The *expression* operand must meet the requirements for an expression as described in subsection 4.7, page 94.

The *attribute* variable specifies a parcel (P), word (W), or value (V) attribute. Attribute, if present, is used rather than the expression's attribute. If a parcel-address attribute is specified, an expression with word-address attribute is multiplied by four; if word-address attribute is specified, an expression with parcel-address attribute is divided by four. An immobile or relocatable expression cannot be specified as having a value attribute.

The following example illustrates the use of the SET pseudo instruction:

```
SIZE      =     o'100
PARAM     SET   D'18
WORD      SET   *W
PARCEL    SET   *P
SIZE      =     SIZE+1      ; Illegal
PARAM     SET   PARAM+2     ; Legal
```

SKIP

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.

You can specify the SKIP pseudo instruction 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.

The format of the SKIP pseudo instruction is as follows:

[*name*]	SKIP	[*count*]

The *name* variable specifies an optional name for a conditional sequence of code. If both *name* and *count* are present, *name* takes precedence. *name* must meet the requirements for identifiers as described in subsection 4.2, page 67.

The *count* variable specifies a statement count. It must be an absolute expression with a 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 used only 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.

The following example illustrates the use of the `SKIP` pseudo instruction:

```
        SKIP                 ; No skipping occurs.
SNAME1  SKIP                 ; Statements are skipped if an ENDIF or
        .                    ; ELSE with a matching location field
        .                    ; label is found.
        .
SNAME1  ENDIF
        .
        .
        .
SNAME2  SKIP  10             ; Statements are skipped until an ENDIF
                             ; or ELSE with a matching location field
        .                    ; label is found.
        .
SNAME2  ENDIF
        .
        .
        .
        SKIP  4              ; Four statements are skipped.
```

SPACE

The `SPACE` pseudo instruction inserts the number of blank lines specified into the output listing. `SPACE` is a list control pseudo instruction and by default, is not listed. To include the `SPACE` pseudo instruction on the listing, specify the `LIS` option on the `LIST` pseudo instruction.

You can specify the `SPACE` pseudo instruction 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.

The format of the `SPACE` pseudo instruction is as follows:

ignored	`SPACE`	[*expression*]

The expression variable specifies an optional absolute expression that specifies 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 1 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.

The *expression* operand must meet the requirement for an expression as described in subsection 4.7, page 94.

STACK

The `STACK` pseudo instruction increases the size of the stack. Increments made by the `STACK` pseudo instruction are cumulative. Each time the `STACK` pseudo instruction 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 instruction.

The `STACK` pseudo instruction is used in conjunction with sections that have a type of `STACK`. If either a `STACK` section or the `STACK` pseudo instruction is specified within a module, the loader tables that the assembler produces 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` pseudo instructions, if any, specified within a module.

You must specify the `STACK` pseudo instruction 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 not recognized as a pseudo instruction.

The format of the `STACK` pseudo instruction is as follows:

ignored	`STACK`	[*expression*]

The *expression* variable is optional. If 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 be defined previously.

If `STACK` is specified without *expression,* the stack is not incremented. The *expression* operand must meet the requirements for an expression as described in subsection 4.7, page 94.

START

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

You must specify the START pseudo instruction from within a program module. If the START pseudo instruction is found within a definition or skipping sequence, it is defined and is not recognized as a pseudo instruction.

The format of the START pseudo instruction is as follows:

ignored	START	*symbol*

The *symbol* variable must be the name of a symbol that is 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 instruction cannot be specified in a section with a type of data only.

The length of the symbol is restricted depending on the type of loader table that the assembler is currently generating. If the symbol is too long, an error message results.

The *symbol* operand must meet the requirements for symbols as described subsection 4.3, page 69.

The following example illustrates the use of the START pseudo instruction:

```
         IDENT   EXAMPLE
         START   HERE
HERE     =       *
         .
         .
         .
         END
```

STOPDUP

The STOPDUP pseudo instruction stops duplication of a code sequence indicated by a DUP or ECHO pseudo instruction.

The STOPDUP pseudo instruction is described in detail in subsection 6.10, page 180.

SUBTITLE

The SUBTITLE pseudo instruction specifies the subtitle that will be printed on the listing. The instruction also causes a page eject. SUBTITLE is a list control pseudo instruction and is, by default, not listed. To include the SUBTITLE pseudo instruction on the listing, specify the LIS option on the LIST pseudo instruction.

You can specify the SUBTITLE pseudo instruction 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.

The format of the SUBTITLE pseudo instruction is as follows:

ignored	SUBTITLE	[*del-char*[*string-of-ASCII*]*del-char*]

The *del-char* variable is the delimiting character. It must be a single matching character on both ends of the ASCII character string. Apostrophes and spaces are not legal delimiters; all other ASCII characters are allowed. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

The *string-of-ASCII* variable is an ASCII character string that will be printed as the subtitle on subsequent pages of the listing. This string replaces any previous string found within the subtitle field.

TEXT

Source lines that follow 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 a reference to the symbol from a listed statement exists. The text *name* part of the cross-reference listing contains the text name.

If the text appears in the global part of a program segment, Symbols defined in *text* source are global. If the text appears within a program module, symbols in *text* source are local.

`TEXT` is a list control pseudo instruction and is, by default, not listed. The `TEXT` pseudo instruction 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.

You can specify the `TEXT` pseudo instruction 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.

The format of the `TEXT` pseudo instruction is as follows:

[*name*]	`TEXT`	[*del-char*[*string-of-ASCII*]*del-char*]

The *name* variable is optional. It is used as the name of the following source 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 it is listed in the text name part of the cross-reference listing.

The *name* location must meet the requirements for names as described in subsection 4.2, page 67.

The *del-char* variable is the delimiting character. It must be a single matching character on both ends of the ASCII character string. Apostrophes and spaces are not legal delimiters; all other ASCII characters are allowed. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

The *string-of-ASCII* variable is an ASCII character string that will be printed as the subtitle on subsequent pages of the listing. A maximum of 72 characters is allowed. This string replaces any previous string found within the subtitle field.

TITLE

The TITLE pseudo instruction specifies the main title that will be printed on the listing. TITLE is a list control pseudo instruction and is, by default, not listed. To include the TITLE pseudo instruction on the listing, specify the LIS option on the LIST pseudo instruction.

You can specify the TITLE pseudo instruction 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.

The format of the TITLE pseudo instruction is as follows:

ignored	TITLE	[*del-char*[*string-of-ASCII*]*del-char*]

The *del-char* variable is the delimiting character. It must be a single matching character on both ends of the ASCII character string. Apostrophes and spaces are not legal delimiters; all other ASCII characters are allowed. Two consecutive occurrences of the delimiting character indicate a single such character will be included in the character string.

The *string-of ASCII* variable is an ASCII character string that will be printed to the diagnostic file. A maximum of 72 characters is allowed.

VWD

The VWD pseudo instruction allows data to be generated in fields that are from 0 to 64 bits wide. Fields can cross word boundaries. Data begins at the current bit position unless a symbol is used in the location field. If a symbol is present within the location field, a forced 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 both. 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.

The format of the `VWD` pseudo instruction is as follows:

```
[symbol]      VWD      [count/[expression]][,[count/[expression]]]
```

The *symbol* variable represents an optional symbol. If *symbol* is 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 subsection 4.3, page 69.

The *count* variable specifies the number of bits in the field. It can be a numeric constant or symbol with absolute and value attributes. *count* must be positive and less than or equal to 64. If a symbol is specified for *count*, it must have been previously defined. If one or more *count* entries are not valid, no code is generated for the entire set of subfields in the operand field; however, each subfield is still evaluated.

The *expression* variable represents the expression whose value will be inserted in the field. If *expression* is missing, the absolute value of 0 is used. If *count* is not equal to 0, 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 subsection 4.7, page 94.

The following example illustrates the use of the `VWD` pseudo instruction:

```
      BASE    M
PDT   BSS     0
      VWD     1/SIGN,3/0,60A'"NAM"'R
                                  ; 1000000000000023440515
                                  ;  10000000653
REMDR =       64-*W               ;  41
      VWD     REMDR/DSN           ;  00011044516
```

In the preceding 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.

User Messages [B]

Text to be supplied later

Character Set [C]

Table 4 lists the character sets supported by CAL.

Table 4. 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
SO	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	14	None
NAK	025/15	3D	None

Table 4. Character set
(continued)

Character	ASCII code (octal/hex)	EBCDIC code (hex)	CDC code (octal)
SYN	026/16	32	None
ETB	027/17	26	None
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

Table 4. Character set
(continued)

Character	ASCII code (octal/hex)	EBCDIC code (hex)	CDC code (octal)
2	062/32	F2	35
3	063/33	F3	36
4	064/34	F4	37
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
;	073/3D	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

Table 4. Character set
(continued)

Character	ASCII code (octal/hex)	EBCDIC code (hex)	CDC code (octal)
N	116/4E	D5	16
O	117/4F	D6	17
P	120/50	D7	20
Q	121/51	D8	21
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
`	140/69	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

Table 4. Character set
(continued)

Character	ASCII code (octal/hex)	EBCDIC code (hex)	CDC code (octal)
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
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

Symbolic Instruction Summary [D]

This appendix provides tables of symbolic machine instructions for Cray PVP systems. See appendix E, page 369, for specific machine applications.

Table 5. Register entry instructions

CAL syntax		Opcode	CAL syntax		Opcode
A*h*	*exp*	01*h*	A*i*	*exp*	020
A*i*	A*i*:*exp*	020	A*i*	*exp*:A*i*	020
A*i*	#*exp*	021	A*i*	−*exp*	021
A*i*	PA*j*	026	A*i*	QA*j*	026
A*i*	−1	031	S*i*	*exp*	040
S*i*	S*i*:*exp*	040	S*i*	*exp*:S*i*	040
S*i*	−*exp*	041	S*i*	#*exp*	041
A*i*	<*exp*	042	S*i*	#>*exp*	042
S*i*	<*exp*	042	A*i*	>*exp*	043
S*i*	>*exp*	043	S*i*	#<*exp*	043
S*i*	SB	051	S*i*	#SB	047
S*i*	0	043	S*i*	1	042
S*i*	−1	042	S*i*	1.	071
S*i*	4.	071	S*i*	2.	071
S*i*	0.4	071	S*i*	0.6	071
SM*jk*	1,TS	003	SM,A*k*	1,TS	003
SM*jk*	0	003	SM,A*k*	0	003
SM*jk*	1	003	SM,A*k*	1	003
V*i*,A*k*	0	077	V*i*	0	145
VL	1	002	VM	0	003
VM0	0	003	VM1	0	003

Table 6. Interregister transfers

CAL syntax		Opcode	CAL syntax		Opcode
A*i*	A*k*	030	A*i*	–A*k*	031
A*i*	CI	033	A*i*	CA,A*j*	033
A*i*	CE,A*j*	033	A*i*	B*jk*	024
A*i*	EA,*j*	023	A*i*	EA,A*j*	023
EA,*j*	A*i*	027	EA,A*j*	A*i*	027
B*jk*	A*i*	025	A*i*	VL	023
VL	A*k*	002	A*i*	SB,A*j*,+1	026
A*i*	SB,+1	026	A*i*	SB,A*j*	026
A*i*	SB*j*	026	SB,A*j*	A*i*	027
SB*j*	A*i*	027	A*i*	S*j*	023
S*i*	A*k*	071	S*i*	+A*k*	071
S*i*	S*k*	051	S*i*	–S*k*	061
S*i*	#S*k*	047	S*i*	+FA*k*	071
S*i*	+FS*k*	062	S*i*	–FS*k*	063
S*i*	T*jk*	074	T*jk*	S*i*	075
S*i*	ST*j*	072	S*i*	ST,A*j*	072
ST*j*	S*i*	073	ST,A*j*	S*i*	073
S*i*	SM	072	SM	S*i*	073
S*i*	SR*j*	073	SR*j*	S*i*	073
S*i*	RT	072	S*i*	VM	073
S*i*	VM0	073	VM	S*j*	003
VM0	A*j*	003	VM1	A*j*	003
VM0	S*j*	003	S*i*	VM1	073
A*i*	VM0	073	A*i*	VM1	073
SETRM	S*i*	073			
VM1	S*j*	003	S*i*	V*j*,A*k*	076
V*i*	CI,S*j*&VM	070	V*i*	0	145
V*i*	–FV*k*	172			

Table 7. Memory transfers

CAL syntax		Opcode	CAL syntax		Opcode
DBM		002	EBM		002
CMR		002	CPA		002
CPR		002	CPW		002
DRI		002	ERI		002
ESC		002	DSC		002
,A0	B*jk*,A*i*	035	B*jk*,A*i*	,A0	034
,A0	T*jk*,A*i*	037	T*jk*,A*i*	,A0	036
0,A0	B*jk*,A*i*	035	B*jk*,A*i*	0,A0	034
0,A0	T*jk*,A*i*	037	T*jk*,A*i*	0,A0	036
A*i*	*exp*,A*h*,BC	10*h*	A*i*	*exp*,A*h*	10*h*
exp,A*h*	A*i*	11*h*	S*i*	*exp*,A*h*,BC	12*h*
exp,A*h*	S*i*	13*h*	S*i*	*exp*,A*h*	12*h*
exp,0	A*i*	110	A*i*	*exp*,0	100
exp,0	S*i*	130	S*i*	*exp*,0	120
exp,	A*i*	110	A*i*	*exp*,	100
exp,	S*i*	130	S*i*	*exp*,	120
,A*h*	A*i*	11*h*	A*i*	,A*h*	10*h*
,A*h*	S*i*	13*h*	S*i*	,A*h*	12*h*
,A0,A*k*	V*j*	177	V*i*	,A0,A*k*	176
,A0,1	V*j*	177	V*i*	,A0,1	176
,A0,V*k*	V*j*	177	V*i*	,A0,V*k*	176
V*i*,V*j*	,A0:A*k*,V*k*	176			

Table 8. Program jumps and exits

CAL syntax		Opcode	CAL syntax		Opcode
J	*exp*	006	J	B*jk*	005
IJ	*exp*	006	IR	*exp*	006
R	*exp*	007	JINV	B*jk*	005
JAZ	*exp*	010	JAN	*exp*	011
JAP	*exp*	012	JAM	*exp*	013
JSZ	*exp*	014	JSN	*exp*	015
JSP	*exp*	016	JSM	*exp*	017
JTS*jk*	*exp*	006	JTS,A*k*	*exp*	006
EX		004	ERR		000

Table 9. Bit count instructions

CAL syntax		Opcode	CAL syntax		Opcode
A*i*	PA*j*	026	A*i*	QA*j*	026
A*i*	PS*j*	026	A*i*	QS*j*	026
A*i*	ZS*j*	027	A*i*	ZA*j*	027
V*i*	PV*j*	174	V*i*	QV*j*	174
V*i*	ZV*j*	175			

Table 10. Shift instructions

CAL syntax		Opcode	CAL syntax		Opcode
S0	Si<exp	052	A0	Ai<exp	052
A0	Ai>exp	053	S0	Si>exp	053
Si	Si<exp	054	Ai	Ai<exp	054
Ai	Ai>exp	055	Si	Si>exp	055
Si	Si,Sj<Ak	056	Si	Si<Ak	056
Si	Si,Sj<1	056	Ai	Ai,Aj<Ak	056
Si	Sj,Si>Ak	057	Si	Sj,Si>1	057
Ai	Ai,Aj>Ak	057	Si	Si>Ak	057
Vi	Vj<Ak	150	Vi	Vj<V0	150
Vi	Vj<1	150	Vi	Vj>Ak	151
Vi	Vj>1	151	Vi	Vj>V0	151
Vi	Vj,Ak	152	Vi	Vj,Vj<1	152
Vi	Vj,Vj<Ak	152	Vi	Vj,Vj>1	153
Vi	Vj,Vj>Ak	153	Vj,[VM]	Vi	153
Vi	Vj,[VM]	153			

Table 11. Integer arithmetic operations

CAL syntax		Opcode	CAL syntax		Opcode
Ai	Aj+Ak	030	Ai	Aj+1	030
Ai	Aj–Ak	031	Ai	Aj–1	031
Ai	Aj*Ak	032			
Si	Sj+Sk	060	Si	Sj–Sk	060
Vi	Sj+Vk	154	Vi	Vj+Vk	155
Vi	Sj–Vk	156	Vi	Vj–Vk	157
Vi	Vj*LVk	165	Vi	Vj*UVk	165
Vi	Sj*LVk	165	Vi	Sj*UVk	165
Vi	Sj*Vk	166	Vi	FLT,Vj	167

Table 12. Floating-point operations

CAL syntax		Opcode	CAL syntax		Opcode
EFI		002	DFI		002
CFP		002			
S*i*	S*j*+FS*k*	062	S*i*	+FS*k*	062
S*i*	S*j*–FS*k*	063	S*i*	–FS*k*	063
S*i*	S*j**FS*k*	064	S*i*	S*j**HS*k*	065
S*i*	S*k*/FS*j*	065	S*i*	S*j**LS*k*	066
S*i*	S*j**US*k*	066	S*i*	S*j**IS*k*	067
S*i*	S*j**RS*k*	066	S*i*	FLT,S*j*	070
	/HS*j*	070	S*i*	SQRT,S*j*	070
S*i*	INT,S*j*	070	S*i*	RINT,S*j*	070
V*i*	S*j**FV*k*	160	V*i*	S*j**HV*k*	162
V*i*	V*j**FV*k*	161	V*i*	V*j**HV*k*	163
V*i*	V*k*/FS*j*	162	V*i*	V*k*/FV*j*	162
S*i*	S*j*,EQ,S*k*	164	S*i*	S*j*,NE,S*k*	164
S*i*	S*j*,GT,S*k*	164	S*i*	S*j*,LE,S*k*	164
S*i*	S*j*,LT,S*k*	164	S*i*	S*j*,GE,S*k*	164
S*i*	S*j*,UN,S*k*	164	VM	S*j*,EQ,V*k*	164
VM	S*j*,NE,V*k*	164	VM	S*j*,GT,V*k*	164
VM	S*j*,LE,V*k*	164	VM	S*j*,LT,V*k*	164
VM	S*j*,GE,V*k*	164	VM	S*j*,UN,V*k*	164
VM	V*j*,EQ,V*k*	164	VM	V*j*,NE,V*k*	164
VM	V*j*,GT,V*k*	164	VM	V*j*,LE,V*k*	164
VM	V*j*,LT,V*k*	164	VM	V*j*,GE,V*k*	164
VM	V*j*,UN,V*k*	164			
V*i*	S*j**RV*k*	164	V*i*	S*j**IV*k*	166
V*i*	V*j**RV*k*	165	V*i*	V*j**IV*k*	167
V*i*	INT,V*j*	167	V*i*	RINT,V*j*	167
V*i*	S*j*+FV*k*	170	V*i*	+FV*k*	170
V*i*	S*j*–FV*k*	172	V*i*	–FV*k*	172

Table 12. Floating-point operations

CAL syntax		Opcode	CAL syntax		Opcode
Vi	Vj+FVk	171	Vi	Vj–FVk	173
Vi	SQRT,Vj	174			
Vi	./HVj	174			

Table 13. Logical operations

CAL syntax		Opcode	CAL syntax		Opcode
Ai	Aj&Ak	044	Si	SB&Sj	044
Si	Sj&Sk	044	Si	Sj&SB	044
Ai	#Ak&Aj	045	Si	#Sk&Sj	045
Si	#SB&Sj	045	Ai	Aj\Ak	046
Si	Sj\Sk	046	Si	Sj\SB	046
Si	SB\Sj	046	Si	#Sj\Sk	047
Ai	#Aj\Ak	047	Si	#SB\Sj	047
Si	#Sj\SB	047	Ai	Aj!Ai&Ak	050
Si	Sj!Si&Sk	050	Si	Sj!Si&SB	050
Ai	Aj!Ak	051	Si	Sj!Sk	051
Si	Sj!SB	051	Si	SB!Sj	051
Vi	Sj&Vk	140	Vi	Sj!Vk	142
Vi	Sj\Vk	144	Vi	Sj!Vk&VM	146
Vi	Vj&Vk	141	Vi	Vj!Vk	143
Vi	Vj\Vk	145	Vi	Vj!Vk&VM	147
Vi	#VM&Vk	146			
VM	Vj,Z	175	VM	Vj,N	175
VM	Vj,P	175	VM	Vj,M	175
Vi,VM	Vj,Z	175	Vi,VM	Vj,N	175
Vi,VM	Vj,P	175	Vi,VM	Vj,M	175

Table 14. Bit matrix multiply instructions

CAL syntax		Opcode	CAL syntax		Opcode
CBL		002	S*i*	S*j**BT	070
BMM	UV*j*	174	BMM	V*j*	174
V*i*	V*j**BT	174			

Table 15. Pass and breakpoint instructions

CAL syntax	Opcode	CAL syntax	Opcode
PASS	001	EBP	002
DBP	002		

Table 16. Monitor operations

CAL syntax		Opcode	CAL syntax		Opcode
CA,A*j*	A*k*	001	CL,A*j*	A*k*	001
CI,A*j*		001	MC,A*j*		001
DI,A*j*		001	EI,A*j*		001
			A*j*	XA	001
XA	A*j*	001	CLN	A*j*	001
BP,*k*	A*j*	001	SIPI	A*j*	001
SIPI		001	CIPI		001
CCI		001	ECI		001
DCI		001	EMI		001
DMI		001	ESI		001
RT	S*j*	001	PCI	S*j*	001
IVC		001	BCD		001
IVCP	A*j*	001	IVCL	A*j*	001

Table 16. Monitor operations

CAL syntax	Opcode	CAL syntax	Opcode
RNM	003	RUM	003
RZM	003	RDM	003

Instructions [E]

This appendix lists symbolic machine instructions for Cray PVP systems. The notes for Table 17 through Table 23 are as follows:

Notes	Meaning
C	Instruction is valid on CRAY C90 systems.
D	Difference in operation between CRAY T90 mode and CRAY C90 mode.
E	Generation depends on the value of *exp.*
F	Functionality of instruction is different on CRAY T90 systems with IEEE floating-point hardware.
I	Instruction is only valid on CRAY T90 systems with IEEE floating-point hardware.
J	Instruction is valid on CRAY J90 systems.
M	Privileged to monitor mode.
N	New instruction.
R	Revised instruction for IEEE floating-point format.
T	Instruction is valid on CRAY T90 systems.
X	Valid only on CRAY Y-MP systems running in X-mode (X-mode not available on CRAY C90 systems).
Y	Instruction is valid on CRAY Y-MP systems running in Y-mode.

Common instructions

F.13

The instructions listed in Table 17 are available on Cray PVP systems as specified in the notes column. If the notes column is empty, the instruction is valid on all Cray PVP systems.

Table 17. Common symbolic machine instructions

Opcode	Notes	CAL		Unit	Function
000000		ERR		–	Error exit.
0010*jk*	M	CA,A*j*	A*k*	–	Sets Current Address (CA) register for channel indicated by (A*j*) to the value specified in (A*k*), activates channel.
001000		PASS		–	Pass instruction.
0011*jk*	M	CL,A*j*	A*k*	–	Sets Channel Limit (CL) register for channel specified by (A*j*) to address specified by (A*k*).
0012*j*0	M	CI,A*j*		–	Clears interrupt flag and error flag for channel specified by (A*j*).
0012*j*1	M	MC,A*j*		–	Clears interrupt and error flags for channel indicated by (A*j*). If (A*j*) represents an output channel, sets device master clear. If (A*j*) represents an input channel, clears device ready-held.
0013*j*0	M	XA	A*j*	–	Enters XA register with (A*j*).
0014*j*0	M	RT	S*j*	–	Loads RTC register with (S*j*).
0014*j*1	M	SIPI	A*j*	–	Sets interprocessor interrupt request to CPU (A*j*); 0≤ (A*j*)≤ 7 on CRAY Y-MP systems.
001401	M	SIPI		–	Sets interprocessor interrupt request of CPU 0.
001402	M	CIPI		–	Clears interprocessor interrupt.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
`0014`*j*`3`	M	`CLN`	`A`*j*	–	Loads Cluster Number (`CLN`) register with (`A`*j*); $0 \leq$ (`A`*j*) $< 9_{10}$ on CRAY Y-MP systems, $0 \leq$ (`A`*j*) $\leq 17_{10}$ on CRAY C90 systems.
`0014`*j*`4`	M	`PCI`	`S`*j*	–	Loads Interrupt Interval (II) register with (`S`*j*).
`001405`	M	`CCI`		–	Clears clock interrupt.
`001406`	M	`ECI`		–	Enables programmable clock interrupt.
`001407`	M	`DCI`		–	Disables programmable clock interrupt.
`001500`	M,C,J, Y			–	Clears all performance monitor counters.
`00200`*k*		`VL`	`A`*k*	–	Transmits (`A`*k*) to `VL` (maximum `VL` = 128 in C90-mode, 64 in Y-mode).
`002000`		`VL`	`1`	–	Enters 1 into `VL`.
`002100`		`EFI`		–	Enables interrupt on floating-point error.
`002200`		`DFI`		–	Disables interrupt on floating-point error.
`002300`		`ERI`		–	Enables interrupt on operand range error.
`002400`		`DRI`		–	Disables interrupt on operand range error.
`002500`		`DBM`		–	Disables bidirectional memory transfers.
`002600`		`EBM`		–	Enables bidirectional memory transfers.
`002700`		`CMR`		–	Completes memory references.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
0030*j*0		VM	S*i*	–	Transmits (S*j*) to VM register.
003000		VM	0	–	Clears VM register.
0034*jk*		SM*jk*	1,TS	–	Tests and sets semaphore *jk*, $0 \le jk \le 37_8$. (Bit 2^2 of *j*=0.)
0036*jk*		SM*jk*	0	–	Clears semaphore *jk*, $0 \le jk \le 37_8$. (bit 2^2 of *j*=0.)
0037*jk*		SM*jk*	1	–	Sets semaphore *jk*, $0 \le jk \le 37_8$. (Bit 2^2 of *j*=0.)
004000		EX		–	Normal exit.
0050*jk*		J	B*jk*	–	Jumps to (B*jk*).
006*ijkm*		J	*exp*	–	Jumps to *exp.*
007*ijkm*	C,J,Y	R	*exp*	–	Return jump to *exp*; set B00 to (*P*)+2.
010*ijkm*	C,J,Y	JAZ	*exp*	–	Jumps to *exp* if (A0)=0.
011*ijkm*	C,J,Y	JAN	*exp*	–	Jumps to *exp* if (A0)≠0.
012*ijkm*	C,J,Y	JAP	*exp*	–	Jumps to *exp* if (A0) positive; includes (A0)=0.
013*ijkm*	C,J,Y	JAM	*exp*	–	Jumps to *exp* if (A0) negative.
014*ijkm*	C,J,Y	JSZ	*exp*	–	Jumps to *exp* if (S0)=0.
015*ijkm*	C,J,Y	JSN	*exp*	–	Jumps to *exp* if (S0)≠0.
016*ijkm*	C,J,Y	JSP	*exp*	–	Jumps to *exp* if (S0) positive; includes (S0)=0.
017*ijkm*	C,J,Y	JSM	*exp*	–	Jumps to *exp* if (S0) negative.
020*i*00*nm*	E	A*i*	*exp*	–	Transmits *exp* to A*i.*
021*i*00*nm*		A*i*	*exp*	–	Transmits ones complement of *exp* to A*i.*

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
022*ijk*	E	A*i*	*exp*	–	Transmits *exp* to A*i*.
023*ij*0		A*i*	S*j*	–	Transmits (S*j*) to A*i*.
023*i*01		A*i*	VL	–	Transmits (VL) to A*i*.
024*ijk*		A*i*	B*jk*	–	Transmits (B*jk*) to A*i*.
025*ijk*		B*jk*	A*i*	–	Transmits (A*i*) to B*jk*.
026*ij*0		A*i*	PS*j*	Pop/LZ	Population count of (S*j*) to A*i*.
026*ij*1		A*i*	QS*j*	Pop/LZ	Population count parity of (S*j*) to A*i*.
026*ij*7		A*i*	SB*j*	–	Transmits (SB*j*) to A*i*.
027*ij*0		A*i*	ZS*j*	Pop/LZ	Leading zero count of (S*j*) to A*i*.
027*ij*7		SB*j*	A*i*	–	Transmits (A*i*) to SB*j*.
030*ijk*		A*i*	A*j*+A*k*	A Int Add	Integer sum of (A*j*) and (A*k*) to A*i*.
030*ij*0		A*i*	A*j*+1	A Int Add	Integer sum of (A*j*) and 1 to A*i*.
030*i*0*k*		A*i*	A*k*	A Int Add	Transmits (A*k*) to A*i*.
031*ijk*		A*i*	A*j*–A*k*	A Int add	Integer difference of (A*j*) less (Ak) to A*i*.
031*ij*0		A*i*	A*j*–1	A Int Add	Integer difference of (A*j*) less 1 to A*i*.
031*i*00		A*i*	–1	A Int Add	Enters –1 into A*i*.
031*i*0*k*		A*i*	–A*k*	A Int Add	Transmits the negative of (A*k*) to A*i*.
032*ijk*		A*i*	A*j**A*k*	A Int Mult	Integer product of (A*j*) and (A*k*) to A*i*.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
033*i*00		A*i*	CI	–	Channel number of highest priority interrupt request to A*i*.
033*ij*0		A*i*	CA,A*j*	–	Address of channel (A*j*) to A*i* (*j*≠0).
033*ij*1		A*i*	CE,Aj	–	Error flag of channel (A*j*) to A*i* (*j*≠0, *k*=1); if C90 mode, include Done flag.
034*ijk*		B*jk*,A*i*	,A0	Memory	Loads (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*.
034*ijk*		B*jk*,A*i*	0,A0	Memory	Loads (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*.
035*ijk*		,A0	B*jk*,A*i*	Memory	Stores (A*i*) words from B registers starting at register *jk* to memory starting at address (A0).
035*ijk*		0,A0	B*jk*,A*i*	Memory	Stores (A*i*) words from B registers starting at register *jk* to memory starting at address (A0).
036*ijk*		T*jk*,A*i*	,A0	Memory	Loads (A*i*) words from memory starting at address (A0) to T registers starting at register *jk*.
036*ijk*		T*jk*,A*i*	0,A0	Memory	Loads (A*i*) words from memory starting at address (A0) to T registers starting at register *jk*.
037*ijk*		,A0	T*jk*,A*i*	Memory	Stores (A*i*) words from T registers starting at register *jk* to memory starting at address (A0).
037*ijk*		0,A0	T*jk*,A*i*	Memory	Stores (A*i*) words from T registers starting at register *jk* to memory starting at address (A0).
040*i*00*nm*		S*i*	*exp*	–	Enters *exp* into S*i*.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
041*i*00*nm*		S*i*	#*exp*	–	Enter ones complement of *exp* into S*i*.
041*i*00*nm*		S*i*	−*exp*	–	Enters twos complement of *exp* into S*i*.
042*ijk*		S*i*	<*exp*	Logical	Forms ones mask in S*i* *exp* bits from the right; *jk* field gets 64 – *exp*.
042*ijk*		S*i*	#>*exp*	Logical	Forms zeros mask in S*i* *exp* bits from the left; *jk* field gets *exp*.
042*i*77		S*i*	1	Logical	Enters 1 into S*i*.
042*i*00		S*i*	−1	Logical	Enters −1 into S*i*.
043*i*00		S*i*	0	Logical	Clears S*i*.
043*ijk*		S*i*	>*exp*	Logical	Forms ones mask in S*i*, *exp* bits from the left; *jk* field gets *exp*.
043*ijk*		S*i*	#<*exp*	Logical	Forms zeros mask in S*i*, *exp* bits from the right; *jk* field gets 64 – *exp*.
044*ijk*		S*i*	S*j*&S*k*	Logical	Logical product of (S*j*) and (S*k*) to S*i*.
044*ij*0		S*i*	S*j*&SB	Logical	Sign bit of (S*j*) to S*i*; *j*≠0.
044*ij*0		S*i*	SB&S*j*	Logical	Sign bit of (S*j*) to S*i*; *j*≠0.
045*ijk*		S*i*	#S*k*&S*j*	Logical	Logical product of (S*j*) and ones complement of (S*k*) to S*i*.
045*ij*0		S*i*	#SB&S*j*	Logical	(S*j*) with sign bit cleared to S*i*.
046*ijk*		S*i*	S*j*\S*k*	Logical	Logical difference of (S*j*) and (S*k*) to S*i*.
046*ij*0		S*i*	S*j*\SB	Logical	Enters (S*j*) into S*i* with sign bit toggled; *j*≠0.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
046*ij*0		S*i*	SB\S*j*	Logical	Enters (S*j*) into S*i* with sign bit toggled; *j*≠0.
047*ijk*		S*i*	#S*j*\S*k*	Logical	Logical equivalence of (S*j*) and (S*k*) to S*i*.
047*ij*0		S*i*	#S*j*\SB	Logical	Logical equivalence of (S*j*) and sign bit to S*i*; *j*≠0.
047*ij*0		Si	#SB\S*j*	Logical	Logical equivalence of sign bit and (S*j*) to S*i*; *j*≠0.
047*i*0*k*		S*i*	#S*k*	Logical	Transmits ones complement of (S*k*) to S*i*.
047*i*00		S*i*	#SB	Logical	Enters ones complement of sign bit into S*i*.
050*ijk*		S*i*	S*j*!S*i*&S*k*	Logical	Scalar merge of (S*i*) and (S*j*) to S*i*.
050*ij*0		S*i*	S*j*!S*i*&SB	Logical	Scalar merge of (S*i*) and sign bit of (S*j*) to S*i*.
051*ijk*		S*i*	S*j*!S*k*	Logical	Logical sum of (S*j*) and (S*k*) to S*i*.
051*ij*0		S*i*	S*j*!SB	Logical	Logical sum of (S*j*) and sign bit to S*i*; *j*≠0.
051*ij*0		S*i*	SB!S*j*	Logical	Logical sum of sign bit and (S*j*) to S*i*; *j*≠0.
051*i*0k		S*i*	S*k*	Logical	Transmits (S*k*) to S*i*.
051*i*00		S*i*	SB	Logical	Enters sign bit into S*i*.
052*ijk*		S0	S*i*<*exp*	Shift	Shifts (S*i*) left *exp* places to S0.
053*ijk*		S0	S*i*>*exp*	Shift	Shifts (S*i*) right *exp* places to S0.
054*ijk*		S*i*	S*i*<*exp*	Shift	Shifts (S*i*) left *exp* places to S*i*.
055*ijk*		S*i*	S*i*>*exp*	Shift	Shifts (S*i*) right *exp* places to S*i*.
056*ijk*		S*i*	S*i*,S*j*<A*k*	Shift	Shifts (S*i*) and (S*j*) left (A*k*) places to S*i*.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
056ij0		Si	Si,Sj<1	Shift	Shifts (Si) and (Sj) left one place to Si.
056i0k		Si	Si<Ak	Shift	Shifts (Si) left (Ak) places to Si.
057ijk		Si	Sj,Si>Ak	Shift	Shifts (Sj) and (Si) right (Ak) places to Si.
057ij0		Si	Sj,Si>1	Shift	Shifts (Sj) and (Si) right one place to Si.
057i0k		Si	Si>Ak	Shift	Shifts (Si) right (Ak) places to Si.
060ijk		Si	Sj+Sk	Int Add	Integer sum of (Sj) and (Sk) to Si.
060ij0	C,Y	Si	Sj+S0	Int Add	Integer sum of (Sj) and the sign bit to Si.
061ijk		Si	Sj–Sk	Int Add	Integer difference of (Sj) less (Sk) to Si.
061ij0	C,Y	Si	Sj–S0	Int Add	Integer difference of (Sj) less the sign bit to Si.
061i0k		Si	–Sk	Int Add	Transmits negative of (Sk) to Si.
062ijk		Si	Sj+FSk	Fp Add	Floating-point sum of (Sj) and (Sk) to Si.
062i0k		Si	+FSk	Fp Add	Normalizes (Sk) to Si.
063ijk		Si	Sj–FSk	Fp Add	Floating-point difference of (Sj) less (Sk) to Si.
063i0k		Si	–FSk	Fp Add	Transmits the negative of (Sk) as a normalized floating-point value to Si.
064ijk		Si	Sj*FSk	Fp Mult	Floating-point product of (Sj) and (Sk) to Si.
065ijk		Si	Sj*HSk	Fp Mult	Half-precision, rounded, floating-point product of (Si) and (Sk) to Si.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
066*ijk*		S*i*	S*j**RS*k*	Fp Mult	Rounded, floating-point product of (S*j*) and (S*k*) to S*i*.
067*ijk*		S*i*	S*j**IS*k*	Fp Mult	2 minus floating-point product of (S*j*) and (S*k*) to S*i*.
070*ij*0		S*i*	/HS*j*	Fp Rcpl	Floating-point reciprocal approximation of (S*j*) to S*i*.
071*i*0*k*		S*i*	A*k*	–	Transmits (A*k*) to S*i* with no sign extension.
071*i*1*k*		S*i*	+A*k*	–	Transmits (A*k*) to S*i* with sign extension.
071*i*2*k*		S*i*	+FA*k*	–	Transmits (A*k*) to S*i* as an unnormalized floating-point value.
071*i*30		S*i*	0.6	–	Transmits 0.75*(2^{48}) to S*i* as a normalized floating-point constant.
071*i*40		S*i*	0.4	–	Transmits 0.5 to S*i* as a normalized floating-point constant.
071*i*50		S*i*	1.	–	Transmits 1.0 to S*i* as a normalized floating-point constant.
071*i*60		S*i*	2.	–	Transmits 2.0 to S*i* as a normalized floating-point constant.
071*i*70		S*i*	4.	–	Transmits 4.0 to S*i* as a normalized floating-point constant.
072*i*00		S*i*	RT	–	Transmits (RTC) to S*i*.
072*i*02		S*i*	SM	–	Transmits semaphores to S*i*.
072*ij*3		S*i*	ST*j*	–	Transmits (ST*j*) register to S*i*.
073*i*00		S*i*	VM	–	Transmits (VM) to S*i*.
073*i*02		SM	S*i*	–	Loads semaphores from S*i*.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
073*i*01		S*i*	SR0	–	Transmits (SR0) to S*i*. This is the only form of the SR*j* syntax valid on CRAY Y-MP systems.
073*i*05		SR0	S*i*	–	Transmits (S*i*) bits 2^{48} through 2^{52} to SR0.
073*i*21	M,C,Y	S*i*	SR2	–	Reads PM counters 00 through 17 and increment pointer. If in Y-mode, increments performance counter.
073*i*31	M,C,Y	S*i*	SR3	–	Reads PM counters 20 through 37 and increment pointer. If in Y-mode, clears all maintenance modes.
074*ijk*		S*i*	T*jk*	–	Transmits (T*jk*) to S*i*.
075*ijk*		T*jk*	S*i*	–	Transmits (S*i*) to T*jk*.
076*ijk*		S*i*	V*j*,A*k*	–	Transmits (V*j*, element (A*k*)) to S*i*.
077*ijk*		V*i*,A*k*	S*j*	–	Transmits (S*j*) to V*i* element (A*k*).
077*i*0*k*		V*i*,A*k*	0	–	Clears element (A*k*) of register V*i*.
100*i*00*nm*		A*i*	*exp*,0	Memory	Loads from (*exp*) to A*i*.
100*i*00*nm*		A*i*	*exp*,	Memory	Loads from (*exp*) to A*i*.
10*hi*0000		A*i*	,A*h*	Memory	Loads from (A*h*) to A*i*.
11*hi*00*nm*		*exp*,A*h*	A*i*	Memory	Stores (A*i*) to (A*h*)+*exp*; A*h*≠0.
110*i*00*nm*		*exp*,0	A*i*	Memory	Stores (A*i*) to *exp*.
110*i*00*nm*		*exp*,	A*i*	Memory	Stores (A*i*) to *exp*.
11*hi*0000		,A*h*	A*i*	Memory	Stores (A*i*) to (A*h*).
12*hi*00*nm*		S*i*	*exp*,A*h*	Memory	Loads from ((A*h*)+*exp*) to S*i*; A*h*≠0.
120*i*00*nm*		S*i*	*exp*,0	Memory	Loads from (*exp*) to S*i*.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
120*i*00*nm*		S*i*	*exp*,	Memory	Loads from (*exp*) to S*i*.
12*hi*0000		S*i*	,A*h*	Memory	Loads from (A*h*) to S*i*.
13*hi*00*nm*		*exp*,A*h*	S*i*	Memory	Stores (S*i*) to (A*h*)+*exp*; A*h*≠0.
130*i*00*nm*		*exp*,0	S*i*	Memory	Stores (S*i*) to *exp*.
130*i*00*nm*		*exp*,	S*i*	Memory	Stores (S*i*) to *exp*.
13*hi*0000		,A*h*	S*i*	Memory	Stores (S*i*) to (A*h*).
140*ijk*		V*i*	S*j*&V*k*	Logical	Logical products of (S*j*) and (V*k*) to V*i*.
141*ijk*		V*i*	V*j*&V*k*	Logical	Logical products of (V*j*) and (V*k*) to V*i*.
142*ijk*		V*i*	S*j*!V*k*	Logical	Logical sums of (S*j*) and (V*k*) to V*i*.
142*i*0*k*		V*i*	V*k*	Logical	Transmits (V*k*) to V*i*.
143*ijk*		V*i*	V*j*!V*k*	Logical	Logical sums of (V*j*) and (V*k*) to V*i*.
144*ijk*		V*i*	S*j*\V*k*	Logical	Logical differences of (S*j*) and (V*k*) to V*i*.
145*ijk*		V*i*	V*j*\V*k*	Logical	Logical differences of (V*j*) and (V*k*) to V*i*.
145*iii*		V*i*	0	Logical	Clears V*i*.
146*ijk*		V*i*	S*j*!V*k*&VM	Logical	Vector merge of (S*j*) and (V*k*) to V*i*.
146*i*0*k*		V*i*	#VM&V*k*	Logical	Vector merge of (V*k*) and zero to V*i*.
147*ijk*		V*i*	V*j*!V*k*&VM	Logical	Vector merge of (V*j*) and (V*k*) to V*i*.
150*ijk*		V*i*	V*j*<A*k*	Shift	Shifts (V*j*) left (A*k*) places to V*i*.
150*ij*0		V*i*	V*j*<1	Shift	Shifts (V*j*) left one place to V*i*.
151*ijk*		V*i*	V*j*>A*k*	Shift	Shifts (V*j*) right (A*k*) places to V*i*.
151*ij*0		V*i*	V*j*>1	Shift	Shifts (V*j*) right one place to V*i*.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
152ijk		Vi	Vj,Vj<Ak	Shift	Double-shifts (Vj) left (Ak) places to Vi.
152ij0		Vi	Vj,Vj<1	Shift	Double-shifts (Vj) left one place to Vi.
153ijk		Vi	Vj,Vj>Ak	Shift	Double-shifts (Vj) right (Ak) places to Vi.
153ij0		Vi	Vj,Vj>1	Shift	Double-shifts (Vj) right one place to Vi.
154ijk		Vi	Sj+Vk	Int Add	Integer sums of (Sj) and (Vk) to Vi.
155ijk		Vi	Vj+Vk	Int Add	Integer sums of (Vj) and (Vk) to Vi.
156ijk		Vi	Sj–Vk	Int Add	Integer differences of (Sj) and (Vk) to Vi.
156i0k		Vi	–Vk	Int Add	Transmits twos complement of (Vk) to Vi
157ijk		Vi	Vj–Vk	Int Add	Integer differences of (Vj) less (Vk) to Vi.
160ijk		Vi	Sj*FVk	Fp Mult	Floating-point products of (Sj) and (Vk) to Vi.
161ijk		Vi	Vj*FVk	Fp Mult	Floating-point products of (Vj) and (Vk) to Vi.
162ijk		Vi	Sj*HVk	Fp Mult	Half-precision, rounded, floating-point products of (Sj) and (Vk) to Vi.
163ijk		Vi	Vj*HVk	Fp Mult	Half-precision, rounded, floating-point products of (Vj) and (Vk) to Vi.
164ijk		Vi	Sj*RVk	Fp Mult	Rounded floating-point products of (Sj) and (Vk) to Vi.
165ijk		Vi	Vj*RVk	Fp Mult	Rounded floating-point products of (Vj) and (Vk) to Vi.

Table 17. Common symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
166*ijk*		V*i*	S*j**V*k*	Fp Mult	32-bit integer product of (S*j*) and (V*k*) to V*i*.
167*ijk*		V*i*	V*j**IV*k*	Fp Mult	Two minus floating-point products of (V*j*) and (V*k*) to V*i*.
170*ijk*		V*i*	S*j*+FV*k*	Fp Add	Floating-point sums of (S*j*) and (V*k*) to V*i*.
170*i*0*k*		V*i*	+FV*k*	Fp Add	Normalizes (V*k*) to V*i*.
171*ijk*		V*i*	V*j*+FV*k*	Fp Add	Floating-point sums of (V*j*) and (V*k*) to V*i*.
172*ijk*		V*i*	S*j*–FV*k*	Fp Add	Floating-point differences of (S*j*) less (V*k*) to V*i*.
172*i*0*k*		V*i*	–FV*k*	Fp Add	Transmits normalized negative of (V*k*) to V*i*.
173*ijk*		V*i*	V*j*–FV*k*	Fp Add	Floating-point differences of (V*j*) less (V*k*) to V*i*.
174*ij*0		V*i*	/HV*j*	Fp Rcpl	Floating-point reciprocal approximations of (V*j*) to V*i*.
174*ij*1		V*i*	PV*j*	V Pop	Population count of (V*j*) to V*i*.
174*ij*2		V*i*	QV*j*	V Pop	Population count parities of (V*j*) to V*i*.
1750*j*0		VM	V*j*,Z	V Logical	Sets VM bits for zero elements of V*j*.
1750*j*1		VM	V*j*,N	V Logical	Sets VM bits for nonzero elements of V*j*.
1750*j*2		VM	V*j*,P	V Logical	Sets VM bits for positive elements of V*j*.
1750*j*3		VM	V*j*,M	V Logical	Sets VM bits for negative elements of V*j*.

Table 17. Common symbolic machine instructions

Opcode	Notes	CAL		Unit	Function
175*ij*4		V*i*,VM	V*j*,Z	V Logical	Sets VM bits for zero elements of V*j*, and register V*i* to the compressed indices of V*j* for zero elements of V*j*.
175*ij*5		V*i*,VM	V*j*,N	V Logical	Sets VM bits for nonzero elements of V*j*, and register V*i* to the compressed indices of V*j* for nonzero elements of V*j*.
175*ij*6		Vi,VM	V*j*,P	V Logical	Sets VM bits for positive elements of V*j*, and register V*i* to the compressed indices of V*j* for positive elements of V*j*.
175*ij*7		V*i*,VM	V*j*,M	V Logical	Sets VM bits for negative elements of V*j*, and register V*i* to the compressed indices of V*j* for negative elements of V*j*.
176*i*0*k*		V*i*	,A0,A*k*	Memory	Loads (VL) words from memory starting at (A0) incrementing by (A*k*) and load into V*i*.
176*i*00		V*i*	,A0,1	Memory	Loads (VL) words from consecutive memory addresses starting with (A0) and load into V*i*.
176*i*1*k*		V*i*	,A0,V*k*	Memory	Reads (VL) words to V*i* from (A0) + (V*k*)
1770*jk*		,A0,A*k*	V*j*	Memory	Stores (VL) words from (V*j*) to memory starting at (A0) incrementing by (A*k*).
1770*j*0		,A0,1	V*j*	Memory	Stores (V*j*) to memory in consecutive addresses starting with (A0).
1771*jk*		,A0,V*k*	V*j*	Memory	Stores (VL) words from V*j* to (A0) + (V*k*).

CRAY J90 and CRAY Y-MP specific instructions

F.14

The instructions listed in Table 18 are available on all CRAY J90 and CRAY Y-MP systems.

Table 18. CRAY J90 and CRAY Y-MP symbolic machine instructions

Opcode	Notes	CAL		Unit	Function
0015*j*0	M			–	Selects performance monitor.
001501	M			–	Disables Port A error correction.
001511	M			–	Disables Port B error correction.
001521	M			–	Disables Port D error correction.
001531	M			–	Enables T register data to be routed through Port D error correction, rather than Port B.
001541	M			–	Enables replacement of check byte with data on Ports C and D writes, and replacement of data with check bytes on Ports A, B, and D reads.
001551	M			–	Enables replacement of check byte with V*k* data on Port C during execution of 1771*jk*.
01*hijkm*	X,E	A*h*	*exp*	–	Transmits *exp* to A*h* (bit 2^2 of *i*=1).
020*ijkm*	X,E	A*i*	*exp*	–	Transmits *exp* to A*i*.
021*ijkm*	X	A*i*	#*exp*	–	Transmits ones complement of *exp* to A*i*.
021*ijkm*	X	A*i*	–*exp*	–	Transmits twos complement of *exp* to A*i*.
040*ijkm*	X	S*i*	*exp*	–	Enters *exp* into S*i*.
041*ijkm*	X	S*i*	#*exp*	–	Enters ones complement of *exp* into S*i*.
041*ijkm*	X	S*i*	–*exp*	–	Enters twos complement of *exp* into S*i*.

Table 18. CRAY J90 and CRAY Y-MP symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
10h*ijkm*	X	A*i*	*exp*,A*h*	Memory	Loads from ((A*h*)+*exp*) to A*i*; A*h*≠0.
100*ijkm*	X	A*i*	*exp*,0	Memory	Loads from (*exp*) to A*i*.
100*ijkm*	X	A*i*	*exp*,	Memory	Loads from (*exp*) to A*i*.
10*hi*000	X	A*i*	,A*h*	Memory	Loads from (A*h*) to A*i*.
10*hi*00*nm*	Y	A*i*	*exp*,A*h*	Memory	Loads A*i* from ((A*h*)+*exp*)
11*hijkm*	X	*exp*,A*h*	A*i*	Memory	Stores (A*i*) to (A*h*)+*exp*; A*h*≠0.
110*ijkm*	X	*exp*,0	A*i*	Memory	Stores (A*i*) to *exp*.
110*ijkm*	X	*exp*,	A*i*	Memory	Stores (A*i*) to *exp*.
11*hi*000	X	,A*h*	A*i*	Memory	Stores (A*i*) to (A*h*).
12*hijkm*	X	S*i*	*exp*,A*h*	Memory	Loads from ((A*h*)+*exp*) to S*i*; A*h*≠0.
120*ijkm*	X	S*i*	*exp*,0	Memory	Loads from (*exp*) to S*i*.
120*ijkm*	X	S*i*	*exp*,	Memory	Loads from (*exp*) to S*i*.
12*hi*000	X	S*i*	,A*h*	Memory	Loads from (A*h*) to S*i*.
13*hijkm*	X	*exp*,A*h*	S*i*	Memory	Stores (S*i*) to (A*h*)+*exp*; A*h*≠0.
130*ijkm*	X	*exp*,0	S*i*	Memory	Stores (S*i*) to *exp*.
130*ijkm*	X	*exp*,	S*i*	Memory	Stores (S*i*) to *exp*.
13*hi*000	X	,A*h*	S*i*	Memory	Stores (S*i*) to (A*h*).
166*ijk*	X	V*i*	S*j**IV*k*	Fp Mult	Two minus floating-point products of (S*j*) and (V*k*) to V*i*.

CRAY C90 specific instructions

F.15

The instructions listed in Table 19 are specific to CRAY C90 systems and CRAY T90 systems running in C90 mode.

Table 19. CRAY C90 symbolic machine instructions

Opcode	Notes	CAL		Unit	Function
0012*j*2	M	DI,A*j*		–	Disables channel (A*j*) interrupts.
0012*j*3	M	EI,A*j*		–	Enables channel (A*j*) interrupts.
001302	M	EMI		–	Enables monitor mode interrupt modes.
001303	M	DMI		–	Disables monitor mode interrupt modes.
001600	M	ESI		–	Enables system I/O interrupts.
0017*jk*	M	BP,*k*	A*j*	–	Transmits (A*j*) to breakpoint address *k*. *k*=0 sets lower-address limit; *k*=1 sets upper-address limit.
002301		EBP		–	Enables interrupt on breakpoint.
002401		DBP		–	Disables interrupt on breakpoint.
002704		CPA		–	Complete port reads and writes.
002705		CPR		–	Completes port reads.
002706		CPW		–	Completes port writes.
0030*j*0		VM0	S*j*	–	Transmits (S*j*) to VM register.
003000		VM0	0	–	Clears VM register.
0030*j*1		VM1	S*j*	–	Transmits (S*j*) to VM upper register.
003001		VM1	0	–	Clears VM1 register.

Table 19. CRAY C90 symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
0034*jk*		SM,A*k*	1,TS	–	Test and set semaphore (A*k*) (Bit 2^2 of *j*=1.)
0036*jk*		SM,A*k*	0	–	Clears semaphore (A*k*). (Bit 2^2 of *j*=1.)
0037*jk*		SM,A*k*	1	–	Sets semaphore (A*k*). (Bit 2^2 of *j*=1.)
00400*k*		EX*k*		–	Exit *k*.
0051*jk*		JINV	B*jk*	–	Jumps to (B*jk*) (maintenance only, invalidate instruction buffers).
006000*nm*		J	*exp*	–	Jumps to *exp.*
0064*jknm*		JTS*jk*	*exp*	–	Jumps to *exp* if SM*jk*=1; else, set SM*jk* = 1 (bit 2^2 of *j*=0).
0064*jknm*		JTS,A*k*	*exp*	–	Jump to *exp* if SM,A*k*=1; else set SM,A*k* = 1 (bit 2^2 of *j*=1).
026*ij*2	D	A*i*	PA*j*	Pop/LZ	Population count of (A*j*) to A*i.*
026*ij*3	D	A*i*	QA*j*	Pop/LZ	Population count parity of (A*j*) to A*i.*
026*ij*4		A*i*	SB,A*j*,+1	–	Transmits (SB) designated by (A*j*) to A*i,* increment by 1.
026*ij*5		A*i*	SB*j*,+1	–	Transmits (SB*j*) to A*i*; increment by 1.
026*ij*6		A*i*	SB,A*j*	–	Transmits (SB) designated by (A*j*) to A*i.*
027*ij*6		SB,A*j*	A*i*	–	Transmits (A*i*) to SB designated by (A*j*).
040*i*20*nm*		S*i*	S*i*:*exp*	–	Transmits *exp* into S*i* bits 2^0-2^{31} bits 2^{32}–2^{63} are unchanged.

Table 19. CRAY C90 symbolic machine instructions

Opcode	Notes	CAL		Unit	Function
040*i*40*nm*		S*i*	*exp*:S*i*	–	Transmits *exp* into S*i* bits 2^{32}-2^{63} bits 2^0–2^{31} are unchanged.
072*ij*6		S*i*	ST,A*j*	–	Transmits (ST) designated by (A*j*) to S*i*.
073*i*00		S*i*	VM0	–	Transmits (VM) to S*i*.
073*i*10		S*i*	VM1	–	Transmits (VM1) to S*i*.
073*ij*1	M	S*i*	SR*j*	–	Transmits (SR*j*) to S*i*.
073*ij*6		ST,A*j*	S*i*	–	Transmits (S*i*) to ST designated by (A*j*).
073*ij*5		SR*j*	S*i*	–	Transmits (S*i*) to SR*j*.
073*i*25	M	SR2	S*i*	–	Issues PM maintenance advance.
073*i*75	M	SR7	S*i*	–	Transmits (S*i*) to maintenance mode register.
10*hi*20*nm*	D	A*i*	*exp*,A*h*,BC	Memory	Loads A*i* from ((A*h*)+*exp*) bypassing data cache and invalidating cache line
005400 150*ij*0		V*i*	V*j*<V0	Shift	Shifts (V*j*) left (V0) places to V*i*.
005400 151*ij*0		V*i*	V*j*>V0	Shift	Shifts (V*j*) right (V0) places to V*i*.
005400 152*ijk*		V*i*	V*j*,A*k*	Shift	Transfers (V*j*) starting at element (A*k*) to (V*i*) starting at element 0.
174*ij*3		V*i*	ZV*j*	V Pop	Leading zero count of (V*j*) to V*i*.

CRAY J90 specific instructions

F.16

The instructions listed in Table 20 are specific to CRAY J90 systems. All of the instructions listed in this table are new instructions.

Table 20. CRAY J90 symbolic machine instructions

Opcode	Notes	CAL	Unit	Function
`0015`*j*`0`	M	N/A	_	Selects performance monitor.
`001501`	M	N/A	–	Disables port A error correction.
`001511`	M	N/A	–	Disables port B error correction.
`001521`	M	N/A	–	Disables port D I/O error correction.
`001541`	M	N/A	–	Enables replacement of checkbyte with data on ports for writes and the replacement of data with checkbytes on ports for reads.
`001551`	M	N/A	–	Replaces checkbits with V*k* data bits on the path to the VA ASIC during execution of instruction `1771`*jk*.
`0016`*j*`1`		`IVC`	–	Send invalidate cache request to CPU (A*j*).

CRAY T90 specific instructions

F.17

The instructions listed in Table 21 are specific to CRAY T90 systems. The instructions listed in Table 19, page 386, are available on CRAY T90 systems running in C90 mode.

Table 21. CRAY T90 symbolic machine instructions

Opcode	Notes	CAL		Unit	Function
0013*j*1	M	A*j*	XA	–	Enters A*j* register with (XA).
001640	M	BCD		–	Broadcasts cluster detach.
0016*j*1	M,N	IVCP	A*j*	–	Invalidate cache in CPU (A*j*).
0016*j*2	M,N	IVCL	A*j*	–	Invalidate cache in CPUs in cluster (A*j*).
002101	N,I	EFI	INV	–	Enables floating-point invalid interrupts.
002102	N,I	EFI	DIV	–	Enables floating-point divide by zero interrupts.
002103	N,I	EFI	OVF	–	Enables floating-point overflow interrupts.
002104	N,I	EFI	UNF	–	Enables floating-point underflow interrupts.
002105	N,I	EFI	INX	–	Enables floating-point inexact interrupts.
002106	N,I	EFI	INP	–	Enables floating-point exceptional input interrupts.
002201	N,I	DFI	INV	–	Disables floating-point invalid interrupts.
002202	N,I	DFI	DIV	–	Disables floating-point divide by zero interrupts.
002203	N,I	DFI	OVF	–	Disables floating-point overflow interrupts.
002204	N,I	DFI	UNF	–	Disables floating-point underflow interrupts.

Table 21. CRAY T90 symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
002205	N,I	DFI	INX	–	Disables floating-point inexact interrupts.
002206	N,I	DFI	INP	–	Disables floating-point exceptional input interrupts.
002501		ESC		–	Enables scalar cache (sets SCE to 1).
002601		DSC		–	Disables and invalidates scalar cache (clears SCE to 0).
002707	N,I	CFP		–	Completes all floating-point operations.
0030*j*2		VM0	A*j*	–	Transmits (A*j*) to VM register.
0030*j*3		VM1	A*j*	–	Transmits (A*j*) to VM upper register.
003004	N,I	RNM		–	Sets round to nearest mode.
003005	N,I	RUM		–	Sets round up mode.
003006	N,I	RZM		–	Sets round to zero mode.
003007	N,I	RDM		–	Sets round down mode.
006100*nm*		IJ	*exp*	–	Jumps to address in *exp.*
007100*nm*		IR	*exp*	–	Return jump to address in *exp*; set B00 to (*P*)+3.
020*i*20*nm*		A*i*	A*i*:*exp*	–	Transmits *exp* to low-order 32 bits of A*i.*
020*i*40*nm*		A*i*	*exp*:A*i*	–	Transmits *exp* to high-order 32 bits of A*i.*
023*ij*6	M	A*i*	EA,*j*	–	Transmits exit address *j* to A*i.*
023*ij*7	M	A*i*	EA,A*j*	–	Transmits exit address (A*j*) to A*i.*
026*ij*2	D	A*i*	PA*j*	Pop/LZ	Population count of (A*j*) to A*i.*

Table 21. CRAY T90 symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
026*ij*3	D	A*i*	QA*j*	Pop/LZ	Population count parity of (A*j*) to A*i*.
027*ij*1		A*i*	ZA*j*	Pop/LZ	Leading zero count of (A*j*) to A*i*.
027*ij*2	M	EA*j*	A*i*	–	Transmits (A*j*) to exit address *j*.
027*ij*3	M	EA,A*j*	A*i*	–	Transmits (A*i*) to exit address (A*j*).
005400 042*ijk*		A*i*	<*exp*	Logical	Forms ones mask in A*i* *exp* bits from the right; *jk* field gets 64 – *exp*.
005400 043*ijk*		A*i*	>*exp*	Logical	Forms ones mask in A*i*, *exp* bits from the left; *jk* field gets *exp*.
005400 044*ijk*		A*i*	A*j*&A*k*	Logical	Logical product of (A*j*) and (A*k*) to A*i*.
005400 045*ijk*		A*i*	#A*k*&A*j*	Logical	Logical product of (A*j*) and (A*k*) to A*i*.
005400 046*ijk*		A*i*	A*j*\A*k*	Logical	Logical difference of (A*j*) and (A*k*) to A*i*.
005400 047*ijk*		A*i*	#A*j*\A*k*	Logical	Logical equivalence of (A*j*) and (A*k*) to A*i*.
005400 050*ijk*		A*i*	A*j*!A*i*&A*k*	Logical	Merge A*i* and A*j* to A*i* by using (A*k*) as mask.
005400 051*ijk*		A*i*	A*j*!A*k*	Logical	Logical sum of (A*j*) and (A*k*) to A*i*.
005400 052*ijk*		A0	A*i*<*exp*	Shift	Shifts (A*i*) left *exp* places to A0.
005400 053*ijk*		A0	A*i*>*exp*	Shift	Shifts (A*i*) right *exp* places to A0.
005400 054*ijk*		A*i*	A*i*<*exp*	Shift	Shifts (A*i*) left *exp* places to A*i*.

Table 21. CRAY T90 symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
005400 055*ijk*		A*i*	A*i*>*exp*	Shift	Shifts (A*i*) right *exp* places to A*i*.
005400 056*ijk*		A*i*	A*i*,A*j*<A*k*	Shift	Shifts (A*i*) and (A*j*) left (A*k*) places to A*i*.
005400 057*ijk*		A*i*	A*j*,A*i*>A*k*	Shift	Shifts (A*j*) and (A*i*) right (A*k*) places to A*i*.
062*ijk*	R	S*i*	S*j*+FS*k*	–	Floating-point S*j* plus S*k* to S*i*.
063*ijk*	R	S*i*	S*j*–FS*k*	–	Floating-point S*j* minus S*k* to S*i*.
064*ijk*	R	S*i*	S*j**FS*k*	–	Floating-point S*j* times S*k* to S*i*.
065*ijk*	F	S*i*	S*j*/FS*j*	–	Floating-point S*k* divided by S*j* to S*i*.
066*ijk*	F	S*i*	S*j**LS*k*	–	Integer S*j* times S*k* to S*i*, returning lower.
005400 066*ijk*	N,I	S*i*	S*j**US*k*	–	Integer S*j* times S*k* to S*i*, returning upper.
070*ij*0	F	S*i*	SQRT,S*j*	–	Floating-point square root of S*j* to S*i*.
070*ij*1		V*i*	CI,S*j*&VM	–	Transmit compressed index of (S*j*) controlled by (VM) to V*i*
070*ij*2	N,I	S*i*	INT,S*j*	–	Floating-point S*j* to integer S*i*.
070*ij*3	N,I	S*i*	RINT,S*j*	–	Floating-point S*j* to rounded integer S*i*.
070*ij*4	N,I	S*i*	FLT,S*j*	–	Integer S*j* to floating-point S*i*.
073*ij*3		ST*j*	S*i*	–	Transmits (S*i*) to ST*j*.
005400 073*i*05	N,I	SETRM	S*i*	–	Set rounding mode from S*i*.
073*i*20		A*i*	VM0	–	Transmits (VM0) to A*i*.

Table 21. CRAY T90 symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
073*i*30		A*i*	VM1	–	Transmits (VM1) to A*i*.
10*hi*20*nm*	D	A*i*	*exp*,A*h*,BC	Memory	Loads A*i* from ((A*h*)+*exp*) bypassing data cache and invalidating cache line
10*hi*40*pnm*		A*i*	*exp*,A*h*	Memory	Loads A*i* from ((A*h*)+*exp*)
10*hi*60*pnm*		A*i*	*exp*,A*h*,BC	Memory	Loads A*i* from ((A*h*)+*exp*) bypassing data cache and invalidating cache line
11*hi*40*pnm*		*exp*,A*h*	A*i*	Memory	Stores (A*i*) to ((A*h*)+*exp*)
12*hi*20*nm*		S*i*	*exp*,A*h*,BC	Memory	Loads S*i* from ((A*h*)+*exp*) bypassing data cache and invalidating cache line
12*hi*40*pnm*		S*i*	*exp*,A*h*	Memory	Loads S*i* from ((A*h*)+*exp*)
12*hi*60*pnm*		S*i*	*exp*,A*h*,BC	Memory	Loads S*i* from ((A*h*)+*exp*) bypassing data cache and invalidating cache line
13*hi*40*pnm*		*exp*,A*h*	S*i*	Memory	Stores (S*i*) to ((A*h*)+*exp*)
005400 153*ij*0		V*i*	V*j*,[VM]	–	Compress V*j* by (VM) to V*i*.
005400 153*ij*1		V*j*,[VM]	V*i*	–	Expand V*j* by [VM] to V*i*.
160*ijk*	R	V*i*	S*j**FV*k*	–	Floating-point S*j* times V*k* to V*i*.
161*ijk*	R	V*i*	V*j**FV*k*	–	Floating-point V*j* times V*k* to V*i*.
162*ijk*	F	V*i*	V*k*/FS*j*	–	Floating-point V*k* divided by S*j* to V*i*.
163*ijk*	F	V*i*	V*k*/FV*j*	–	Floating-point V*k* divided by V*j* to V*i*.
005501 164*ijk*	N,I	S*i*	S*j*,EQ,S*k*	–	Floating-point compare equal.

Table 21. CRAY T90 symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
005502 164*ijk*	N,I	S*i*	S*j*,NE,S*k*	–	Floating-point compare not equal.
005503 164*ijk*	N,I	S*i*	S*j*,GT,S*k*	–	Floating-point compare greater than.
005504 164*ijk*	N,I	S*i*	S*j*,LE,S*k*	–	Floating-point compare less than or equal.
005505 164*ijk*	N,I	S*i*	S*j*,LT,S*k*	–	Floating-point compare less than.
005506 164*ijk*	N,I	S*i*	S*j*,GE,S*k*	–	Floating-point compare greater than or equal.
005507 164*ijk*	N,I	S*i*	S*j*,UN,S*k*	–	Floating-point compare unordered.
005521 1640*jk*	N,I	VM	S*j*,EQ,V*k*	–	Floating-point compare equal.
005522 1640*jk*	N,I	VM	S*j*,NE,V*k*	–	Floating-point compare not equal.
005523 1640*jk*	N,I	VM	S*j*,GT,V*k*	–	Floating-point compare greater than.
005524 1640*jk*	N,I	VM	S*j*,LE,V*k*	–	Floating-point compare less than or equal.
005525 1640*jk*	N,I	VM	S*j*,LT,V*k*	–	Floating-point compare less than.
005526 1640*jk*	N,I	VM	S*j*,GE,V*k*	–	Floating-point compare greater than or equal.
005527 1640*jk*	N,I	VM	S*j*,UN,V*k*	–	Floating-point compare unordered.
005541 1640*jk*	N,I	VM	V*j*,EQ,V*k*	–	Floating-point compare equal.
005542 1640*jk*	N,I	VM	V*j*,NE,V*k*	–	Floating-point compare not equal.
005543 1640*jk*	N,I	VM	V*j*,GT,V*k*	–	Floating-point compare greater than.

Table 21. CRAY T90 symbolic machine instructions (continued)

Opcode	Notes	CAL		Unit	Function
005544 1640*jk*	N,I	VM	V*j*,LE,V*k*	–	Floating-point compare less than or equal.
005545 1640*jk*	N,I	VM	V*j*,LT,V*k*	–	Floating-point compare less than.
005546 1640*jk*	N,I	VM	V*j*,GE,V*k*	–	Floating-point compare greater than or equal.
005547 1640*jk*	N,I	VM	V*j*,UN,V*k*	–	Floating-point compare unordered.
165*ijk*	F	V*i*	V*j**LV*k*	–	Integer V*j* times V*k* to V*i* returning lower.
005400 165*ijk*	N,I	V*i*	V*j**UV*k*	–	Integer V*j* times V*k* to V*i* returning upper.
166*ijk*	F	V*i*	S*j**LV*k*	–	Integer S*j* times V*k* to V*i* returning lower.
005400 166*ijk*	N,I	V*i*	S*j**UV*k*	–	Integer S*j* times V*k* to V*i* returning upper.
167*ij*0	F	V*i*	INT,V*j*	–	Floating-point V*j* to integer V*i*.
167*ij*1	F	V*i*	RINT,V*j*	–	Floating-point V*j* to rounded integer V*i*.
167*ij*2	F	V*i*	FLT,V*j*	–	Integer V*j* to floating-point V*i*.
170*ijk*	R	V*i*	S*j*+FV*k*	–	Floating-point S*j* plus V*k* to V*i*.
171*ijk*	R	V*i*	V*j*+FV*k*	–	Floating-point V*j* plus V*k* to V*i*.
172*ijk*	R	V*i*	S*j*–FV*k*	–	Floating-point S*j* minus V*k* to V*i*.
173*ijk*	R	V*i*	V*j*–FV*k*	–	Floating-point V*j* minus V*k* to V*i*.

Table 21. CRAY T90 symbolic machine instructions

Opcode	Notes	CAL		Unit	Function
174*ij*0	F	V*i*	SQRT,V*j*	–	Floating-point square root of V*j* to V*i*.
005400 176*ijk*		V*i* :V*j*	,A0:A*k*,V*k*	Memory	Loads V*i* from memory using addresses (A0) + (V*k*) and load V*j* from memory using addresses (A*k*) + (V*k*).

Bit Matrix multiply instructions

F.18

The instructions listed in Table 22 are available only on systems that support the bit matrix multiply (BMM) function.

Table 22. Bit matrix multiply symbolic machine instructions

Opcode	Notes	CAL		Unit	Function
002210		CBL		–	Clears the B matrix loaded bit in the exchange package and the status register.
070*ij*6	N	S*i*	S*j**BT	BMM	Load single bit matrix element S*j* into the BMM functional unit. Generate results of S*j**BT and store the results in S*i*. S*j* must be left-justified and zero-filled.
1740*j*5	N	BMM	UV*j*	–	Transmits V*j* elements 64 through 127 to B matrix.

Table 22. Bit matrix multiply symbolic machine instructions

Opcode	Notes	CAL		Unit	Function
174*ij*4		BMM	V*j*	BMM	Load elements 0 through VL of V*j* into the BMM functional unit as B^t. Matrix B must be stored in V*j*.
174*ij*6		V*i*	V*j**BT	BMM	Logical bit matrix multiply of V*j* and elements 0 through VL of matrix B^t to V*i*. Matrix A must be left-justified and zero-filled in V*j*; result matrix C is stored in V*i* left-justified to bit 63.

Special register values and logical operators

F.19

Table 23 shows special register values and logical operators.

Table 23. Special register values and logical operators

Register	Value	Logical operators	
A*h*, *h*=0	0		0101
A*i*, *i*=0	(A0)		1100
A*j*, *j*=0	0	(&, AND, Product)	0100
A*k*, *k*=0	1		
S*i*, *i*=0	(S0)		0101
S*j*, *j*=0	0		1100
S*k*, *k*=0	2^{63}	(!, OR, Sum)	1101
			0101
			1100
		(\, XOR, Difference)	1001