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:

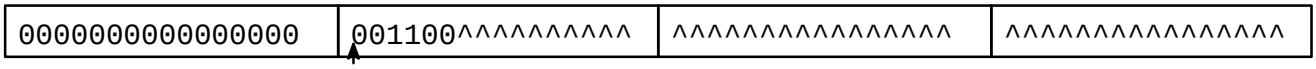

Figure 30. `BITP` example – resetting the pointer

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
```

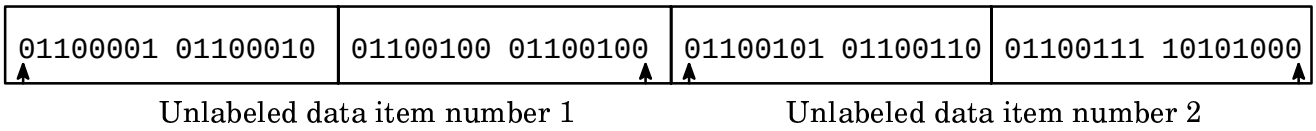

Figure 32. Storage of unlabeled data items

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.
```

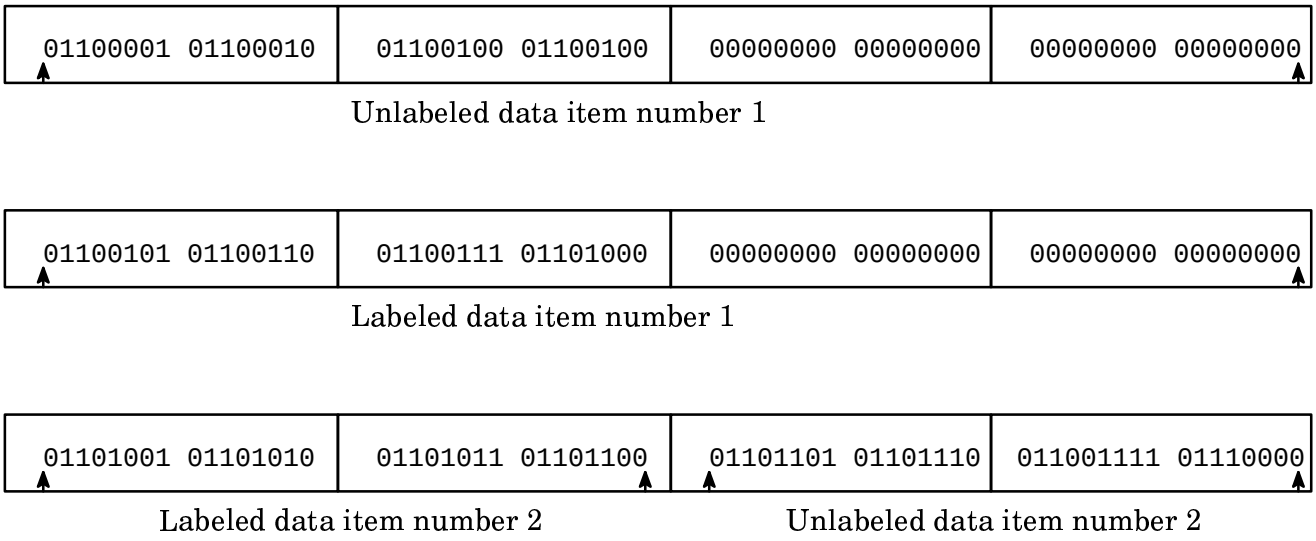

Figure 33. Storage of labeled and unlabeled data items

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.
```

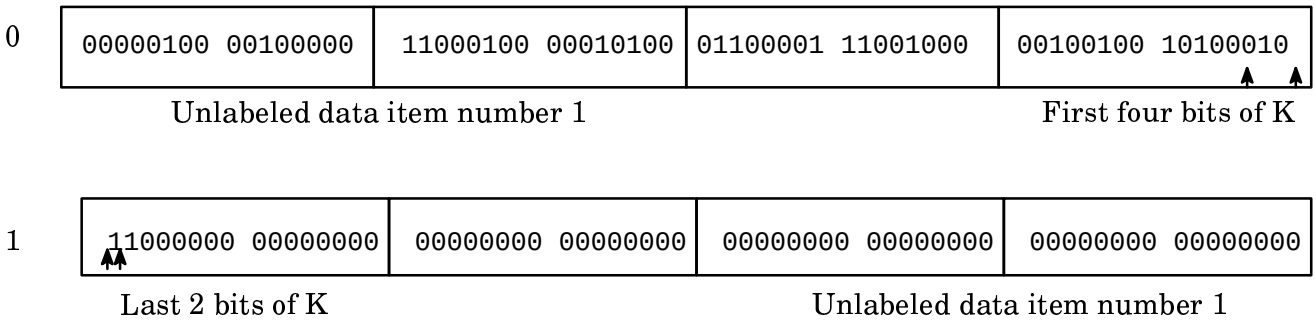

Figure 34. Storage of CDC character data item

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         ; 00000000000000000000
                               ;  040502206420
      DATA   'EFGHIJ'*         ;             10521443510
                               ;  044512
LL2   DATA   'ABCD'            ; 0405022064204010020040
      DATA   100               ; 00000000000000000144
      DATA   1.25E-9           ; 0377435274616704302142

      DATA   'THIS IS A MESSAGE'*L
                               ; 0521102225144022251440
                               ; O4044O2324252324640507
                               ;  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      [expression1][,[expression2[,[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,l0,ZERO  ; Decimal string with zero fill.
_*      "THREE"                 ; Minus sign, zero fill, value.
*       -000256000              ; Minus sign, zero fill, value.
FOUR    DECMIC —256000,l0,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'l00=,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      [expression1] [",[expression2[",[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,l0,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   sectioncode         ; 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   sectioncommon,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.