The CAL Program [3]

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

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

The following subsections describe each of these components.

Program segment
3.1

A CAL program consists of zero or more segments. A CAL program with zero segments consists of one or more empty files. A file that contains one blank line is considered a segment. For example, CAL considers a program with an `IDENT/END` sequence that is followed by a blank line to contain two segments. Ordinarily, each segment consists of global definitions, a program module, or a combination of global definitions and a program module. Figure 5, page 34, illustrates the organization of a CAL program.

Program module
3.1.1

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

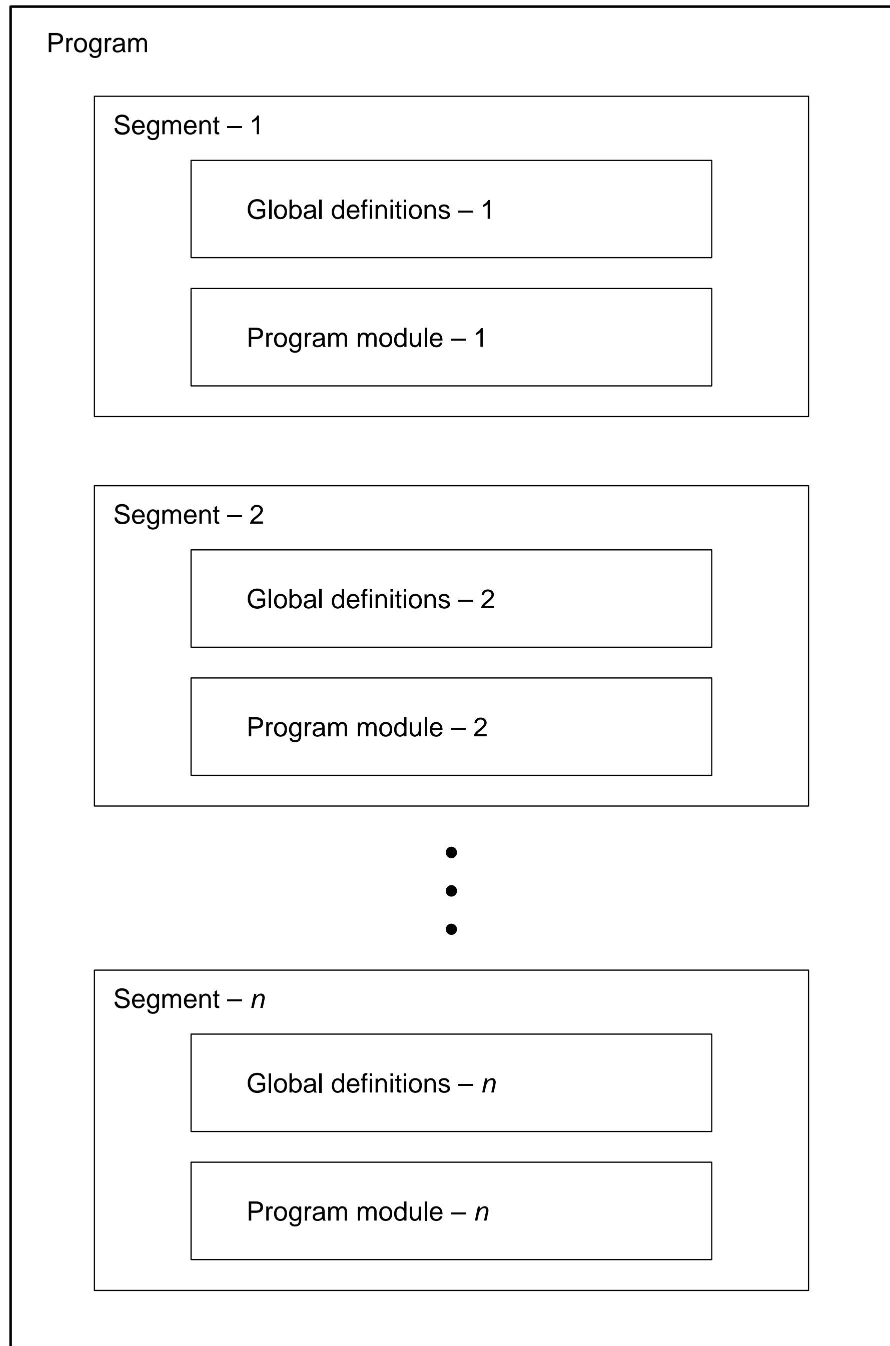

Figure 5. CAL program organization

Global definitions 3.1.2

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

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

Redefinable micros, redefinable symbols, and symbols of the form %%*x*; where *x* is 0 or more identifier-characters are exceptions. Although they can occur in such sequences, they are local to the segment in which they are defined, are not known to the assembler after the next END pseudo instruction (end of the current segment) is encountered, and they are not included in the cross-reference listing. Symbols defined within the global definitions area cannot be qualified (see subsection 4.3.1, page 70).

The following example illustrates global definitions:

```
SYM1   =    1           ; Begin segment 1 global
                        ; SYM1 cannot be redefined
SYM2   SET  2           ; SYM2 equals 2 for this module
%%SYM3 =    3           ; Gone at the end of the module
%%SYM4 SET  4           ; Gone at the end of the module

       IDENT  TEST1     ; Beginning of module 1
       S1   SYM1        ; Register S1 gets 1
       S2   SYM2        ; Register S2 gets 2
       S3   %%SYM3      ; Register S3 gets 3
       S4   %%SYM4      ; Register S4 gets 4
       END              ; End of segment 1 and module TEST 1

SYM2   SET  3           ; Beginning of segment 2
%%SYM3 =    5           ; Global definitions
       IDENT  TEST2     ; Beginning of module TEST 2
       S1   SYM1        ; Register S1 gets 1
       S2   SYM2        ; Register S2 gets 3
       S3   %%SYM3      ; Register S3 gets 5
       S4   %%SYM4      ; Error:  not defined
       END              ; End of segment 2 and module TEST 2

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

Source statement
3.2

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

Formal parameters, symbols, names, pseudo instructions, and macro names are case-sensitive. To be recognized, subsequent references to a previously defined formal parameter, symbol, name, or functional unit must match the original definition character-for-character and case-for-case (uppercase or lowercase).

The following are examples of case-sensitivity:

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

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

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

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

New format 3.2.1

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

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

- Location
- Result
- Operand
- Comment

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

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

Location field 3.2.1.1

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

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

Result field 3.2.1.2

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

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

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

Operand field 3.2.1.3

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

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

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

Comment field 3.2.1.4

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

The following example illustrates the use of the comment field:

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

Old format
3.2.2

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

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

- Location
- Result
- Operand
- Comment

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

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

Location field
3.2.2.1

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

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

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

Result field
3.2.2.2

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

Operand field
3.2.2.3

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

Comment field
3.2.2.4

The comment field is optional and begins with the first nonblank character following the operand field or, if the operand field is empty, does not begin before column 35. If the result field extends beyond column 32 and no operand entry is provided, two or more blanks must precede the comment field. The comment field can be the only field supplied in a statement. If editing is enabled, comments are edited. For more information about editing, see subsection 3.3, page 41.

The following example illustrates the use of the comment field:

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

Statement editing
3.3

CAL processes source statements sequentially from the source file. Statement editing is a form of preprocessing in which CAL deletes or replaces characters before processing the statement as source code.

The assembler performs the following types of statement editing:

- Concatenation

  The assembler recursively deletes all underscore characters and combines the character that preceded the underscore with the character following the underscore.

- Micro substitution

  The assembler replaces a micro name with a predefined character string. The character string replacement is not edited a second time.

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

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

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

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

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

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

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

is edited as follows:

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

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

Micro substitution 3.3.1

You can assign a micro name to a character string. You can refer to that character string in subsequent statements by its micro name. The CAL assembler searches for quotation marks (") that delimit micro names. The first quotation mark indicates the beginning of a micro name; the second quotation mark identifies the end of a micro name. Before a statement is interpreted, CAL replaces the micro name with the character string that comprises the micro. For more information on micros, see subsection 3.5, page 48.

Concatenate 3.3.2

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

Append 3.3.3

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

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

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

Continuation 3.3.4

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

Comment 3.3.5

A semicolon (;) in any column (new format only) or an asterisk (*) in column 1 indicates a comment line. The assembler lists comment lines, but they have no effect on the program. When a semicolon or an asterisk has an editing symbol after it, the symbol is treated as part of the comment and is not used. In the new format, comment statements with semicolons or asterisks are not appended.

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

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

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

The macro in the preceding example is expanded as follows:

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

Actual statements and edited statements
3.3.6

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

This following example shows an actual statement:

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

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

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

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

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

Instructions
3.4

CAL recognizes two types of instructions:

- Assembler-defined
- User-defined

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

Assembler-defined instructions
3.4.1

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

- Machine instructions
- Pseudo instructions

Machine instructions 3.4.1.1

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

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

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

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

Pseudo instructions 3.4.1.2

Pseudo instructions direct the assembler in its task of interpreting the source statements and generating an object program. CAL has a large complement of pseudo instructions. Each pseudo instruction has a unique identifier in the result field. The contents of the location and operand fields depend on the pseudo instruction.

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

User-defined instructions 3.4.2

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

CAL recognizes four types of defined sequences: macro, opdef, dup, and echo. Defined sequences are classified as either permanent or temporary.

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

Micros
3.5

Through the use of micros, you can assign a name to a character string and subsequently refer to the character string by its name. A reference to a micro results in the character string being substituted for the name before assembly of the source statement containing the reference. The `CMICRO`, `MICRO`, `OCTMIC`, and `DECMIC` pseudo instructions (described in subsection 5.10, page 124) assign the name to the character string.

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

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

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

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

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

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

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

```
EQUALTAG   S0              S1    ; The location of S0 and S1 on the
                                 ; source statement is unchanged
```

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

```
"PFX"TAG   S0              S1    ; Because LESS is one character shorter
                                 ; than the micro string name "PFX", the
                                 ; values in the result and operand
                                 ; fields are shifted one space to the
                                 ; left.
```

Before the line is interpreted, CAL substitutes `LESS` for `"PFX"`, producing the following line:

```
LESSTAG  S0             S1    ; Because LESS is one character shorter
                              ; than the micro string name "PFX", the
                              ; values in the result and operand
                              ; fields are shifted one space to the
                              ; left.
```

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

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

Before the line is interpreted, CAL substitutes the predefined string `greater` for `"pfx"`. Because the predefined substitute string is 2 characters longer than micro name, the fields to the right of the substitution are shifted 2 characters to the right, producing the following statement:

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

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

{*microname*}

When a micro that contains one or more embedded micros is encountered, CAL edits all embedded micros within the micro until a micro name is recognized or until the micro name is determined to be illegal (undefined or exceeding the maximum allowable string length of 8 characters). When an illegal micro is encountered, CAL issues an appropriate message and terminates the editing of the micro. An embedded micro also can contain one or more embedded micros.

The following example includes valid and not valid defined embedded micros

```
index    micro          \1\   ; Assigns literal value to index
null     micro          \\    ; Assigns literal value to null

array "index"  micro \Some string\
array1 micro \Some string\†

* "array1" - an explicit reference.
* Some string - an explicit reference†
* "array" "index" - not valid, because "array" was not defined.
* "array"1 - not valid, because "array" was not defined.†
* "array{index}" - This is an example of an embedded micro.
* Some string - This is an example of an embedded micro.†
* "{null}array{index}" - This is an example of two embedded micros.
* Some string - This is an example of two embedded micros.†
```

† Edited by CAL

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

```
index    micro           \1\   ; Assigns literal value to index
null     micro           \\    ; Assigns literal value to null

array"index" micro \Some string\
array1 micro \Some string\†

* "{nu{n{null}u{null}ll}ll}ar{null{null}}ray{ind{null}ex}" - Micro
* Some string - Micro†
```

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

† Edited by CAL

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

```
identity   micro         \The substitute string for this example\
null       micro         \\    ; Assigns literal value to null

* "identity{null}" - This is a valid micro.
* The substitute string for this example - This is a valid micro.†

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

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

Sections
3.6

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

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

† Edited by CAL

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

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

Local sections 3.6.1

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

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

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

Main sections 3.6.1.1

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

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

Literals section 3.6.1.2

The first use of a literal value in an expression causes the assembler to store the data item in a literals section. Data is generated in the literals section implicitly by the occurrence of a literal. Explicit data generation or memory reservation is not allowed in the literals section. The assembler supports the literals section as a constant section. For more information about literals, see subsection 4.4.3, page 85.

Sections defined by the SECTION *pseudo instruction* 3.6.1.3

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

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

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

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

A section with the same name, type, and location used in different areas of a program is recognized as the same section. For more information, see the SECTION pseudo instruction in appendix A, page 189.

Common sections 3.6.2

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

At program end, each common section is identified to the loader by its SECTION name and is available for reference by another program module. If you specify the ORG pseudo instruction, an exception to these conditions can occur. Specifying the ORG pseudo instruction may cause the placement of code or memory reservations to be different from the currently specified working section.

If a common section is specified, the identifier in the location field that names the section must be unique within the module in which it is defined. When a section is assigned a type (COMMON, DYNAMIC, ZEROCOM, or TASKCOM) that differs from the type of a previously defined section, it cannot be assigned the name of a previously defined section within the same module.

Section stack buffer 3.6.3

CAL maintains a stack buffer that contains a list of the sections specified. Each time a SECTION pseudo instruction names a new section, CAL adds the name of the section to the list and identifies the new section as the current section. You also can use the BLOCK and COMMON pseudo instructions to name sections.

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

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

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

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

```
      IDENT     STACK          ; The IDENT statement puts the first entry
                               ; on the list of qualifiers.  This entry
                               ; starts the symbol table for unqualified
                               ; symbols.
SYM1  =         1              ; SYM1 is relative to the main section.
      QUAL      QNAME1         ; Second entry on the list of qualifiers.
SYM2  =         2              ; SYM2 is the first entry in the symbol
                               ; table for QNAME1.
SNAME SECTION   MIXED          ; SNAME is the second entry on the list of
                               ; sections
      MLEVEL    ERROR          ; Reset message level to error eliminate
                               ; warning level messages.
SYM3  =         *              ; SYM3 is the second entry in the symbol
                               ; table for QNAME1 and is relative to the
                               ; SNAME section.
      MLEVEL    *              ; Reset message level to default in effect
                               ; before the MLEVEL specification.
      SECTION   *              ; SNAME is deleted from the list of
                               ; sections.
SYM4  =         4              ; SYM4 is the third entry in the symbol
                               ; table for QNAME1 and is relative to the
                               ; main section.
      QUAL      QNAME2         ; Third entry on the list of qualifiers.
SYM5  =         5              ; SYM5 is the first entry in the symbol
                               ; table for QNAME2.
SYM6  =         /QNAME1/SYM2   ; SYM6 gets SYM2 from the symbol table for
                               ; QNAME1 even though QNAME1 is not the
                               ; current qualifier in effect.
      QUAL      *              ; QNAME2 is removed as the current
                               ; qualifier name.
SYM7  =         6              ; SYM7 is the fourth entry in the symbol
                               ; table for QNAME1.
SYM8  =         7              ; Second entry in the symbol table for
                               ; unqualified symbols.
```

Origin counter
3.6.3.1

The *origin counter* controls the relative location of the next word that will be assembled or reserved in the section. You can reserve blank memory areas by using either the ORG or BSS pseudo instructions to advance the origin counter.

When the special element *O is used in an expression, the assembler replaces it with the current parcel-address value of the origin counter for the section in use. To obtain the word-address value of the origin counter, use W.*O. For more information about the special elements and W. prefix, see subsection 4.5, page 89.

Location counter
3.6.3.2

Usually, the *location counter* is the same value as the origin counter and the assembler uses it to define symbolic addresses within a section. The counter is incremented when the origin counter is incremented. Use the LOC pseudo instruction to adjust the location counter so that it differs in value from the origin counter or so that it refers to the address relative to a section other than the one currently in use. When the special element * is used in an expression, the assembler replaces it with the current parcel-address value of the location counter for the section in use. To obtain the word-address value of the location counter, Use W.* (see subsection 4.5, page 89).

Word-bit-position counter
3.6.3.3

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

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

Force word boundary 3.6.3.4

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

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

Parcel-bit-position counter 3.6.3.5

In addition to the word-bit-position counter, CAL also maintains a counter that points to the next bit to be assembled in the current parcel. This pointer is the *parcel-bit-position counter*. It is 0 at the beginning of a new parcel and advances by 1 for each completed bit in the parcel. The maximum value is 15 for the rightmost bit in a parcel. When a parcel is completed, the parcel-bit-position counter is reset to 0.

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

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

Force parcel boundary 3.6.3.6

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