Defined Sequences [6]

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

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

The defined sequence pseudo instructions are as follows:

Pseudo	Description
MACRO	A sequence of source program instructions saved by the assembler for inclusion in a program when called for by the macro name. The macro call resembles a pseudo instruction.
OPDEF	A sequence of source program instructions saved by the assembler for inclusion in a program called for by the OPDEF pseudo instruction. The opdef resembles a symbolic machine instruction.
DUP	Introduces a sequence of code that is assembled repetitively a specified number of times; the duplicated code immediately follows the DUP pseudo instruction.
ECHO	Introduces a sequence of code that is assembled repetitively until an argument list is exhausted.
ENDM	Ends a macro or opdef definition.
ENDDUP	Terminates a DUP or ECHO sequence of code.
STOPDUP	Stops the duplication of a code sequence by overriding the repetition condition.

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

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

Similarities among defined sequences
6.1

Defined sequences have the following functional similarities:

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

Editing
6.1.1

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

If editing is enabled, editing of the definition occurs during assembly each time it is called. The ENDDUP, ENDM, END, INCLUDE, and LOCAL pseudo instructions and prototype statements should not contain micros or concatenation characters because they may not be recognized at definition time.

When a sequence is defined, editing is disabled and cannot be explicitly enabled. When a sequence is called, CAL performs the following operations:

- Checks all parameter substitutions marked at definition time
- Edits the statement if editing is enabled
- Processes the statement

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

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

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

Definition format 6.1.2

MACRO, OPDEF, DUP, and ECHO pseudo instructions use the same definition format. The format consists of a header, body, and end.

The header consists of a MACRO, OPDEF, DUP, or ECHO pseudo instruction, a prototype statement for a MACRO or OPDEF definition, and, optionally, LOCAL pseudo instructions. For a macro, the prototype statement provides a macro functional

definition and a list of formal parameters. For an opdef, the prototype statement supplies the syntax and the formal parameters.

LOCAL pseudo instructions identify parameter names that CAL must make unique to the assembly each time the definition sequence is placed in a program segment. Asterisk comments can be placed in the header and do not affect the way CAL scans the header. Asterisk comments are dropped from the definition. To force asterisk comments into a definition, see subsection 3.3.5, page 44.

The body of the definition begins with the first statement following the header. The body can consist of a series of CAL instructions other than an END pseudo instruction. The body of a definition can be empty, or it can include other definitions and calls. A definition used within another definition is not recognized, however, until the definition in which it is contained is called; therefore, an inner definition cannot be called before the outer definition is called for the first time.

A comment statement identified by an asterisk in column l is ignored in the definition header and the definition body. Such comments are not saved as a part of the definition sequence. Comment fields on other statements in the body of a definition are saved.

An ENDM pseudo instruction with the proper name in the location field ends a macro or opdef definition. A statement count or an ENDDUP pseudo instruction with the proper name in the location field ends a dup definition. An ENDDUP pseudo instruction with the proper name in the location field ends an echo definition.

Formal parameters 6.1.3

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

- Positional
- Keyword
- Echo
- Local

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

You can embed a formal parameter name within the definition body; however, embedded parameters must satisfy the following requirements:

- The first character of an embedded parameter must begin with a legal initial-identifier-character.
- An embedded parameter cannot be preceded by an initial-identifier-character (for example, PARAM is a legally embedded parameter within the ABC_PARAM_DEF string because it is preceded by an underscore character). PARAM is not a legally embedded character within the string ABCPARAMDEF because it is preceded by an initial-identifier-character (C).
- An embedded parameter must not be followed by an identifier-character.

  In the following example, the embedded parameter is legal because it is followed by an element separator (blank character):

  ```
  PARAM678
  ```

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

  ```
  PARAM6789
  ```

- Embedded parameters must contain 8 or less characters. PARAM6789 is illegal because it contains 9 characters. The character that follows an embedded parameter (9) cannot be an identifier-character.
- If and only if the new format is specified, an embedded parameter must occur before the first comment character (;) of each statement within the body.
- An embedded parameter must have a matching formal parameter name in the definition header.

- Formal parameter names should not be `END`, `ENDM`, `ENDDUP`, `LOCAL`, or `INCLUDE` pseudo instructions. If any of these are used as parameter names, substitution of actual arguments occurs when these names are contained in any inner definition reference.

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

Instruction calls
6.1.4

Each time a definition sequence of code is called, an entry is added to a list of currently active defined sequences within the assembler. The most recent entry indicates the current source of statements to be assembled. When a definition is called within a definition sequence that is being assembled, another entry is made to the list of defined sequences, and assembly continues with the new definition sequence belonging to the inner, or nested, call.

At the end of a definition sequence, the most recent list entry is removed and assembly continues with the previous list entry. When the list of defined sequences is exhausted, assembly continues with statements from the source file.

An inner nested call can be recursive; that is, it can reference the same definition that is referenced by an outer call. The depth of nested calls permitted by CAL is limited only by the amount of memory available.

The sequence field in the right margin of the listing shows the definition name and nesting depth for defined sequences being assembled. Nesting depth numbers begin in column 89 and can be one of the following: `:1`, `:2`, `:3`, `:4`, `:5`, `:6`, `:7`, `:8`, `:9`, `:*`.

If the nesting depth is greater than 9, CAL keeps track of the current nesting level and an asterisk represents nesting depths of 10 or more. Nesting depth numbers are restricted to two characters so that only the two rightmost character positions are overwritten.

If the sequence field (columns 73 through 90) of the source file is not empty, CAL copies the existing field for a call into every statement expanded by the call reserving columns 89 and 90 for the nesting level. For example, if the sequence field for MCALL was LQ5992A.112, the sequence field for a statement expanded from MCALL would read as follows:

```
LQ5992A.1l2     :l
```

Additional nested calls within MCALL would change the nesting level, but the sequence field would be unchanged during MCALL. For example:

```
LQ5992A.112     :2
LQ5992A.l12     :2
LQ5992A.112     :2
LQ5992A.112     :3
LQ5992A.112     :*
LQ5992A.112     :l
```

If the sequence field (columns 73 through 90) of the source file is empty, CAL inserts the name of the definition, as follows:

Name	Description
Macro	The inserted name in the sequence field is the functional found in the result field of the macro prototype statement.
Opdef	The inserted name in the sequence field is the name used in the location field of the OPDEF pseudo instruction itself.
Dup	The inserted name in the sequence field is the name used in the location field of the DUP pseudo instruction, or if the count is specified and name is not, the name is *Dup.
Echo	The inserted name in the sequence field is the name used in the location field of the ECHO pseudo instruction.

In all cases, the first two columns of the sequence field contain asterisks (**) to indicate CAL has generated the sequence field. Columns 89 and 90 of the sequence field are reserved for the nesting level. If, for example, the sequence field is missing for MCALL, it would read as follows:

```
** MCALL   :1
```

Additional nested calls within MCALL would change the nesting level, but the sequence field number would be unchanged for the duration of MCALL.

The following example illustrates how CAL tracks the nesting sequence:

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

Interaction with the INCLUDE *pseudo instruction*
6.1.5

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

Sequence	Description
MACRO	INCLUDE pseudo instructions are expanded at definition time.
OPDEF	INCLUDE pseudo instructions are expanded at definition time.
DUP	INCLUDE pseudo instructions are expanded at definition time. If count is specified, the INCLUDE pseudo instruction statement itself is not included in the statements being counted.
ECHO	INCLUDE pseudo instructions are expanded at definition time.

Macros (MACRO)
6.2

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

You can specify the MACRO pseudo instruction anywhere within a program segment. If the MACRO pseudo instruction is found within a definition, it is defined. If the MACRO pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

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

The following example illustrates a macro definition:

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

Macro definition 6.2.1

The macro definition header consists of the MACRO pseudo instruction, a prototype statement, and optional LOCAL pseudo instructions. The prototype statement provides a name for the macro and a list of formal parameters and default arguments.

A comment statement, identified by an asterisk in column 1, is ignored in the definition header or definition body. Such comments are not saved as a part of the definition sequence. Comment fields on other statements in the body of a definition are saved.

The end of a macro definition is signaled by an ENDM pseudo instruction with a functional name that matches the functional name in the result field of the macro prototype statement. For a description of the ENDM pseudo instruction, see subsection 6.6, page 177.

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

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

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

The format of the macro definition is as follows:

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

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

- *location*

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

- *functional*

  The *functional* variable specifies the name of the macro. It must be a valid identifier or the equal sign. If *functional* is the same as a currently defined pseudo instruction or macro, this definition redefines the operation associated with *functional*, and a message is issued.

- *parameters*

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

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

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

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

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

– !/*

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

A left parenthesis signals the beginning of an *embedded argument* and must be terminated by a matching right parenthesis. An embedded argument can contain an argument or pairs of matching left and right parentheses. If an asterisk precedes the positional parameter name, the embedded argument is used in its entirety. If an asterisk does not precede the positional parameter name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

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

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

A *syntactically valid expression* can include a legal terminator (space, tab, or semicolon for new format) or an element separator (comma). The syntactically valid expression satisfies the requirements for an expression, but it is used only as an argument and is not evaluated in the macro call itself. If the syntactically valid expression is an embedded argument, then, as long as an asterisk precedes the *positional-parameter* name, the embedded argument is used in its entirety. If an asterisk does not precede the *positional-parameter* name, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument. Use of the syntactically valid expression permits you to enter a string (=‘,'R) of characters that may contain one or more spaces or a comma.

The *null string* is an empty string.

– *positional-parameters*

positional-parameters must be specified with valid and unique names and they must meet the requirements for names as described in subsection 4.2, page 67. There can be none, one, or more positional parameters. The default argument for a *positional-parameter* is an empty string.

The positional parameters defined in the macro definition are case-sensitive. Positional parameters that are specified in the definition body must identically match positional parameters defined by the macro prototype statement.

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

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

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

– *keyword-parameters*

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

There can be zero, one, or more *keyword-parameters*. You can enter names within *keyword-parameters* in any order. Default arguments can be provided for each *keyword-parameter* at definition time, and they are used if the keyword is not specified at call time.

The *keyword-parameters* defined in a macro definition are case-sensitive. The *keyword-parameters* specified in the macro body must match the *positional-parameters* specified in the macro prototype statement.

The `!` is optional. If the `!` is not included, the `-s` option argument can be an embedded argument, a character string, or a null string. If the `!` is included, the parameter be either a syntactically valid expression or a null string.

Embedded argument. A left parenthesis signals the beginning of an embedded argument and it must be matched by a right parenthesis. An embedded argument can also contain pairs of matching left and right parentheses. If an asterisk precedes the positional parameter name, the embedded argument is used in its entirety; otherwise, the outermost parentheses are stripped from the embedded argument and the remaining string is used as the argument.

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

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

The null argument is an empty string.

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

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

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

The following example illustrates the use of positional parameters:

```
         MACRO
         JUSTIFY         !PARAM
         .                    ; Macro prototype.
         .
         .
JUSTIFY  ENDM
         JUSTIFY         ','R  ; Macro call
         JUSTIFY         '  'R; Macro call
```

When the following macro is called, the positional parameter `p1` receives a value of `v1` because an asterisk does not precede the parameter on the prototype statement. The positional parameter `p2`, however, receives a value of `(v2)` because an asterisk precedes the parameter on the prototype statement.

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

Macro calls 6.2.2

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

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

The elements of the macro call are described as follows:

- *locarg*

  The *locarg* element specifies an optional location field argument. *locarg* must be terminated by a space or a tab (new format only). *locarg* can be any character up to but not including a space. If a location field parameter is specified on the macro definition, you can specify a matching location field parameter on the macro call. *locarg* is substituted wherever the location field parameter occurs in the definition. If no location field parameter is specified in the definition, this field must be empty.

- *functional*

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

- *positional-arguments*

  Positional-arguments specify an actual argument string that corresponds to a *positional-parameter* that is specified in the definition prototype statement. The requirements for *positional-arguments* are specified by the corresponding *positional-parameter* in the macro definition prototype statement. *Positional-arguments* are not case-sensitive to *positional-parameters* on the macro call.

  The first *positional-argument* is substituted for the first *positional-parameter* in the prototype operand field, the second *positional-argument* string is substituted for the second *positional-parameter* in the prototype operand field, and so on. If the number of *positional-arguments* is less than the number of *positional-parameters* in the prototype operand field, null argument strings are used for the missing *positional-arguments*.

Two consecutive commas indicate a null (empty) *positional-argument* string.

- *keyword-arguments*

  keyword-arguments are an actual argument string that corresponds to a *keyword-parameter* specified in the macro definition prototype statement. The requirements for *keyword-arguments* are specified by the corresponding *keyword-parameter* in the macro definition prototype statement.

  keyword-arguments are not recognized until after *n* subfields (*n* commas); *n* is the number of positional parameters in the operand field of the macro definition.

  You can list *keyword-arguments* in any order; matching the order in which *keyword-parameters* are listed on the macro prototype statement is unnecessary. However, because the *keyword-parameter* is case-sensitive, it must be specified in the macro call exactly as specified in the macro prototype statement to be recognized.

  The default *keyword-parameters* specified in the macro prototype statement are used as the actual *keyword-arguments* for missing *keyword-arguments*.

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

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

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

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

Call and expansion of the INTCONV macro:

```
      INTCONV           A1,S3 ; Macro call.
      S2                +FA1  ; Transfer with special expression and
                              ; sign extension.
      S2                +FS2  ; Normalize the S register.
```

Note: Comments preceded by an underscore and an asterisk are included in the definition bodies of the following macro examples. These comments are included to illustrate the way in which parameters are passed from the macro call to the macro definition. Because comments are not assembled, _* comments allow arguments to be shown without regard to hardware differences or available machine instructions.

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

The macro table contains positional and keyword parameters.

```
        macro
        table           tabn,val1=#0,val2=,val3=0
tables  section         data
tabn    con             'tabn'1
        con             val1
        con             val2
        con             val3
        section         *     : Resume use of previous section.
table   endm
        list            mac
```

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

```
        table           taba,val3=4,val2=a    ; Macro call.
tables  section         data
taba    con             'taba'1
        con             #0
        con             a
        con             4
        section         *     : Resume use of previous section.
```

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

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

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

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

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

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

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

```
       ONE              SYM1  ; Call using one parameter.
* PARAMETER 1:          SYM1  ; SYM1 corresponds to PARAM1.
* PARAMETER 2:                ; Null string.
* PARAMETER 3:                ; Null string.
       macro
       two              param1,param2,param3
_* Parameter 1:         param1
                              ; SYM1 corresponds to param1.
_* Parameter 2:         param2
                              ; SYM2 corresponds to param2.
_* Parameter 3:         param3
                              ; Null string.
two    endm
       list             mac
```

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

```
       two    sym1,sym2       ; Call using two parameters.
* Parameter 1:           sym1  ; sym1 corresponds to param1.
* Parameter 2:           sym2  ; sym2 corresponds to param2.
* Parameter 3:                 ; Null string.
       MACRO
       THREE  PARAM1,PARAM2,PARAM3
_*PARAMETER 1:           PARAM1
                               ;SYM1 corresponds to PARAM1.
_*PARAMETER 2:           PARAM2
                               ;SYM2 corresponds to PARAM2.
_*PARAMETER 3:           PARAM3
                               ;SYM3 corresponds to PARAM3.
THREE  ENDM
       LIST   MAC
```

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

```
       THREE  SYM1,SYM2,SYM3  ; Call matching prototype.
* PARAMETER 1:           SYM1  ; SYM1 corresponds to PARAM1.
* PARAMETER 2:           SYM2  ; SYM2 corresponds to PARAM2.
* PARAMETER 3:           SYM3  ; SYM3 corresponds to PARAM3.
```

The following examples demonstrate the use of the optional !.

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

```
       MACRO
       BANG   PARAMl,!PARAM2,!PARAM3=,PARAM4=
_* PARAMETER 1:          PARAM1
                               ; Embedded argument.
_* PARAMETER 2:          PARAM2
                               ; Syntactically valid expression
_* PARAMETER 3:          PARAM3
                               ; Syntactically valid expression
_* PARAMETER 4:          PARAM4
                               ; Null string.
BANG   ENDM
       LIST   MAC
```

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

```
       BANG   (1,2),'abc,def',PARAM3=1+2
                               ; Macro call.
* PARAMETER 1:           1,2   ; Embedded argument.
* PARAMETER 2:           'abc,def'
                               ; Syntactically valid expression.
* PARAMETER 3:           1+2   ; Syntactically valid expression.
* PARAMETER 4:                 ; Null string.
```

In the previous example:

- If the argument for PARAM1 had been (((1,2))), S1 would have received ((1,2)) at expansion.
- The ! specified on PARAM2 and PARAM3 permits commas and spaces to be embedded within strings 'abc,def' and allows expressions to be expanded without evaluation 1+2.
- PARAM4 passes a null string. A space or comma following the equal sign specifies a null or empty character string as the default argument.

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

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

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

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

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

`NULL` demonstrates the effect of a null string on macro expansions. `P2` is passed a null string. When `NULL` is expanded, the resulting line is left-shifted two spaces, which is the difference between the length of the parameter (`P2`) and the null string.

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

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

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

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

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

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

```
         nullparm
=        1
```

Note: The location field parameter was omitted on the macro call in the previous example. The result and operand fields of the first line of the expansion were shifted left 8 character positions because a null argument was substituted for the 8-character parameter, `LONGPARM`.

If the old format is used, only one space appears between the location field parameter and result field in the macro definition. If a null argument is substituted for the location parameter, the result field is shifted into the location field in column 2. Therefore, at least two spaces should always appear between a parameter in the location field and the first character in the result field in a definition.

If the new format is used, the result field is never shifted into the location field.

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

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

The following illustrates calls and expansions of the DEFAULT macro:

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

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

Macro WRONG shows the incorrect way to specify a literal string in a macro definition. The comments in the expansion are writer comments and are not part of the expansion.

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

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

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

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

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

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

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

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

Macro BAD demonstrates the wrong method of replacing parameters.

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

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

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

Macro good demonstrates the correct method for replacing parameters.

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

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

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

Macro ALPHA demonstrates the specification of an embedded parameter.

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

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

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

CAL processes the embedded parameter in macro ALPHA, as follows:

1. CAL scans the string to identify the parameter. ABC_ cannot be a parameter because the underscore character is not defined as an identifier character for a parameter.
2. CAL identifies PARAM as the parameter when the second underscore character is encountered.
3. 1 is substituted for PARAM, producing string ABC_1_DEFG.
4. If editing is enabled, the underscore characters are removed and the resulting string is ABClDEFG.

   If editing is disabled, the string is ABC_1_DEFG.
5. CAL processes the statement.

Operation definitions (OPDEF) 6.3

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

Opdefs resemble machine instructions and can be used to define new machine instructions or to redefine current machine instructions. Machine instructions map into opcodes that represent some hardware operation. When an operation is required that is not available through the hardware, an opdef can be written to perform that operation. When the opdef is called, the opdef maps into the opdef definition body and the operation is performed by the defined sequence specified in the definition body.

You can replace any existing CAL machine instruction with an opdef. Although opdef definitions should conform to meaningful operations that are supported by the hardware, they are not restricted to such operations.

The opdef definition sets up the parameters into which the arguments specified in the opdef call are substituted. Opdef parameters are always expressed in terms of registers or expressions. The opdef call passes arguments to the parameters in the opdef definition. The syntax for the opdef definition and the opdef call are identical with two exceptions:

- The complex register has been redefined for the opdef definition prototype statement as follows:

```
register_mnemonic.register_parameter
```

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

```
@[expression_parameter]
```

These two exceptions allow you to specify parameters in the place of registers and expressions for an opdef definition.

The syntax defining a *register_parameter* and an *expression_parameter* is case-sensitive. Every character that identifies the parameter in the opdef prototype statement must be identical to every character in the body of the opdef definition. This includes the case (uppercase, lowercase, or mixed case) of each character.

Because the opdef can accept arguments in many forms, it can be more flexible than a macro. Opdefs place a greater responsibility for parsing arguments on the assembler. When a macro is specified, the responsibility for parsing arguments is placed on the user in many cases. Parsing a macro argument can involve numerous micro substitutions, which greatly increase the number of statements required to perform a similar operation with an opdef.

Defined sequences (macros, opdefs, dups, and echos) are costly in terms of assembler efficiency. As the number of statements in a defined sequence increases, the speed of the assembler decreases. This decrease in speed is directly related to the number of statements expanded and the number of times a defined sequence is called.

Limiting the number of statements in a defined sequence improves the performance of the assembler. In some cases, an opdef can perform the same operation as a macro and use fewer statements in the process.

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

```
MANYCALL   OPDEF
           A.REG1        A.REG2!A.REG3
                               ; Opdef prototype statement.
           S1            A.REG2
           S2            A.REG3
           S3            S1!S2
           A.REG1        S3    ; OR of registers S1 and S2.
MANYCALL   ENDM                ; End of opdef definition.
```

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

```
        A1      A2!A3           ; First call to opdef MANYCALL.
        S1      A.2
        S2      A.3
        S3      S1!S2
        A.1     S3              ; OR of registers S1 and S2.
        A.1     A.2!A.3         ; Second call to opdef MANYCALL.
        S1      A.2
        S2      A.3
        S3      S1!S2
        A.1     S3              ; OR of registers S1 and S2.
ONE     =       1               ; Define symbols.
TWO     =       2
THREE   =       3
        A.ONE   A.TWO!A.THREE   ; Third call to opdef MANYCALL.
        S1      A.2
        S2      A.3
        S3      S1!S2
        A.ONE   S3              ; OR of registers S1 and S2.
        A1      A.2!A.THREE     ; Fourth call to opdef MANYCALL.
        S1      A.2
        S2      A.3
        S3      S1!S2
        A.1     s3              ; OR of registers S1 and S2.
```

In the first and second calls to opdef MANYCALL, the arguments passed to REG1, REG2, and REG3 are 1, 2, and 3, respectively. In the third call to opdef MANYCALL, the arguments passed to REG1, REG2, and REG3 are ONE, TWO, and THREE, respectively. The fourth call to opdef MANYCALL demonstrates that the form of the arguments can vary within one call to an opdef if they take a valid form. The arguments passed to REG1, REG2, and REG3 in the fourth call are 1, 2, and THREE, respectively.

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

```
      MACRO
      $IF      REG1,COND,REG2  ; Macro prototype statement.
      .
      .
      .
$IF   ENDM                     ; End of macro definition.
      .
      .
      .
      $IF      S6,EQ,S.3       ; Macro call.
      .
      .
      .
      $ELSE
      .
      .
      .
      $ENDIF
```

Parsing the parameters (`S6,EQ,S3`) passed to the definition requires many micro substitutions within the definition body. These micros increase the number of statements within the definition body.

The same function is performed in the following example, but an opdef is specified instead of a macro. In this instance, specifying an opdef rather than a macro reduces the number of statements required for the function.

Because an opdef is called by its form, it is more flexible than a macro in accepting arguments. The opdef expects to be passed two S registers and the `EQ` mnemonic. You can specify the arguments for the registers in a number of ways and still be recognized as S register arguments by the opdef.

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

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

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

If an opdef occurs within the global definitions part of a program segment, it is defined as global. Opdef definitions are local if they occur within a program module (an IDENT, END sequence). A global definition can be redefined locally, but the global definition is reenabled and the local definition is discarded at the end of the program module. You can reference a global definition anywhere within an assembler program after it has been defined.

You can specify the OPDEF pseudo instruction anywhere within a program segment. If the OPDEF pseudo instruction is found within a definition, it is defined. If the OPDEF pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

In the following example, the operand and comment fields of the expanded line are shifted two positions to the left (difference between reg and 1):

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

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

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

Opdef definition
6.3.1

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

Opdef syntax is uniquely defined on the result field alone, in which case, the operand field is not specified or on the result and operand fields. The OPDEF prototype permits up to three subfields within the result and operand fields. At least one field must be present within the result field. No fields are required in the operand field.

The syntax for each of the subfields within the result and operand fields of the opdef prototype statement is identical. No special syntax forms exist for any of the subfields. The rules that apply for the first subfield in the result field apply to the remainder of the subfields within the result field and to all subfields within the operand field.

The format of the opdef definition is as follows:

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

The variables in the opdef definition are described as follows:

- *name*

  name identifies the opdef definition and has no association with functionals that appear in the result field of instructions. *name* must match the name in the location field of the ENDM pseudo instruction, which ends the definition.

- *loc*

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

- *defsynres*

  defsynres specifies the definition syntax for the result field. It can be one, two, or three subfields specifying a valid result field syntax. The result field must be a symbolic.

  Valid result subfields for opdefs can be one of the following:

  – Initial register
  – Mnemonic
  – Initial expression

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

[*prefix*] [*register–prefix*] *register*[*register-separator*[*register-ending*]]
[*prefix*] [*register-prefix*] *register*[*register-expression-separator* [*register-ending*]]
[*prefix*] [*register-prefix*] *register*[*register-expression-separator* [*expression-ending*]]
[*prefix*] [*register-prefix*] *register*[*special-register-separator* [*register-ending*]]

  The elements of an initial register definition are as follows:

  – *prefix*

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

– *register-prefix*

register-prefix is optional and case-sensitive. When a *register-prefix* is specified on an opdef call, it is recognized by the opdef definition without regard to the case (uppercase or lowercase) in which it was entered. It can be specified as any of the following characters:

```
<       >       #<      #>      #       #F      #f      #H
#h      #I      #i      #P      #p      #Q      #q      #R
#r      #Z      #z      +       +F      +f      +H      +h
+I      +i      +P      +p      +Q      +q      +R      +r
+Z      +z      -       -F      -f      -H      -h      -I
-i      -P      -p″     -Q      -q      -R      -r      -Z
-z      *       *F      *f      *H      *h      *I      *i
*P      *p      *Q      *q      *R      *r      *Z      *z
/       /F      /f      /H      /h      /I      /i      /P
/p      /Q      /q      /R      /r      /Z      /z      F
f       H       h       I       i       P       p       Q
q       R       r       Z       z
```

– *register*

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

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

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

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

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

When you specify a *simple register* or a *complex register* mnemonic on an opdef call, it is recognized by the opdef definition without regard to the case (uppercase, lowercase, or mixed case) in which it was entered.

– *register-separator*

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

```
+F    +f    +H    +h    +I    +i    +P    +p
+Q    +q    +R    +r    +Z    +z
-F    -f    -H    -h    -I    -i    -P    -p
-Q    -q    -R    -r    -Z    -z
*F    *f    *H    *h    *I    *i    *P    *p
*Q    *q    *R    *r    *Z    *z
/F    /f    /H    /h    /I    /i    /P    /p
/Q    /q    /R    /r    /Z    /z
```

– *register-expression-separator*

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

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

– *special-register-separator*

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

– *register-ending*

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

register$_1$ [*register-separator* [*register*$_2$ [*suffix*]]]
register [*register-expression-separator* [*register-or-expression* [*suffix*]]]
register$_1$ [*special-register-separator* [*register*$_2$ [*suffix*]]]

The *register*$_1$, *register-separator*, *register*$_2$, *suffix*, *register*, and *register-expression-separator* elements are described previously under *initial-register*.

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

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

special-register-separator is specified as #.

– *expression-ending*

expression-ending is specified as follows:

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

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

expression-parameter

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

expression-separator can be one of the following:

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

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

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

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

[*prefixl* [*expression-prefix*] *expression* [*expression-separator* [*register-ending*]]
[*prefix*] [*expression-prefix*] *expression* [*expression-separator* [*expression-ending*]]
expression [*expression-separator* [*register-ending*]]
expression [*expression-separator* [*expression-ending*]]

The elements of the initial expression are described as follows:

– *prefix*

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

– *expression-prefix*

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

<, >, #<, or #>

– *expression*

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

expression-parameter

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

– *expression-separator*

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

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

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

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

- *defsynop*

Definition syntax for the operand field; can be zero, one, or two subfields specifying a valid operand field syntax. If a subfield exists in the result field, the first subfield in the operand field must be a symbolic.

The definition syntax for the operand field of an opdef is the same as the definition syntax for the result field of an opdef. See the definition of *defsynres,* earlier in this subsection.

Opdef calls
6.3.2

An opdef definition is called by an instruction that matches the syntax of the result and operand fields as specified in the opdef prototype statement.

The arguments on the opdef call are passed to the parameters on the opdef prototype statement. The special syntax for registers and expressions that was required on the opdef definition does not extend to the opdef call.

The format of the opdef call is as follows:

```
locarg      callsynres      callsynop
```

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

- *locarg*

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

  If a location field parameter is specified on the opdef definition, a matching location field parameter can be specified on the opdef call. *locarg* is substituted wherever the location field parameter occurs in the definition. If no location field parameter is specified in the definition, this field must be empty.

- *callsynres*

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

  The syntax of the result field call is the same as the syntax of the result field definition with two exceptions. The special syntax rules that are in effect for registers and expressions on the opdef definition do not apply to the opdef call. The syntax for registers and expressions used on the opdef call is the same as the syntax for registers and expressions.

The subfields in the result field on the opdef call can be specified with one of the following:

– Initial-register

– Mnemonic

– Initial-expression

For a description of the syntax for the result field of the opdef call, see the syntax for the result field of the opdef definition.

- *callsynop*

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

  The syntax of the operand field call is the same as the syntax of the operand field definition with two exceptions. The special syntax rules that are in effect for registers and expressions on the opdef definition do not apply to the opdef call. The syntax for registers and expressions used on the opdef call is the same as the syntax for registers and expressions.

  The subfields in the operand field on the opdef call can be specified with one of the following:

  – Initial-register

  – Mnemonic

  – Initial-expression

  For a description of the syntax for the operand field of the opdef call, see the syntax for the result field of the opdef definition.

The following rules apply for opdef calls:

- The character strings *callsynres* and *callsynop* must be exactly as specified in the opdef definition.

- An expression must appear whenever an expression in the form @*exp* is indicated in the prototype statement. The actual argument string is substituted in the definition sequence wherever the corresponding formal parameter @*exp* occurs.

- The actual argument string consisting of a *complex-register* mnemonic followed by a period (`.`) followed by a *register-parameter*. A *register-designator* followed by a *register-parameter* must appear wherever the *register-designator* `A.`*register-parameter*, `B.`*register-parameter*, `SB.`*register-parameter*, `S.`*register-parameter*, `T.`*register-parameter*, `ST.`*register-parameter,* `SM.`*register-parameter*, or `V.`*register-parameter*, respectively, appeared in the prototype statement.

  – If the *register-parameter* is of the form *octal-integer*, the actual argument is the *octal-integer* part. The *octal-integer* is restricted to 4 octal digits.

  – If the *register-parameter* is of the form *.integer-constant* or *.symbol*, the actual argument is an *integer-constant* or a symbol.

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

```
fdv  opdef                    ; Scalar floating-point divide prototype
                              ; statement.
L    s.r1     s.r2/fs.r3
     errif    r1,eq,r2
     errif    r1,eq,r3
L    s.r1     /hs.r3
     s.r2     s.r2*fs.r1
     s.r3     s.r3*is.r1
     s.r1     s.r2*fs.r3
fdv  endm
```

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

```
a    s4     s3/fs2            ; Divide s3 by s2, result to s4.
     errif  4,eq,3
     errif  4,eq,2
a    s.4    /hs.2
     s.3    s.3*fs.4
     s.2    s.2*is.4
     s.4    s.3*fs.2
```

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

```
JEQ   OPDEF
L     JEQ      A.A1,A.A2,@TAG  ; Opdef prototype statement.
L     A0       A_A1-A_A2
_*    JAZ      @TAG            ; Expression is expected.
JEQ   ENDM                     ; End of opdef definition.
      LIST     MAC             ; Listing expansion.
```

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

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

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

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

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

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

The following example demonstrates how the expansion of an opdef is affected when the opdef call does not include a label that was specified in the opdef definition:

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

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

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

The location field parameter was omitted on the opdef call in the previous example. The result and operand fields of the first line of the expansion were shifted left three character positions because a null argument was substituted for the 3-character parameter, `lbl`.

If the old format is used, only one space appears between the location field parameter and result field in the macro definition. If a null argument is substituted for the location parameter, the result field is shifted into the location field in column 2. Therefore, at least two spaces should always appear between a parameter in the location field and the first character in the result field in a definition.

If the new format is used, the result field is never shifted into the location field.

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

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

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

```
       S1   #pa2              ; Recognized by CASE.
       S1   #Pa2              ; Recognized by CASE.
       S1   #pA2              ; Recognized by CASE.
       S1   #PA2              ; Recognized by CASE.
       s1   #pa2              ; Recognized by CASE.
       s1   #Pa2              ; Recognized by CASE.
       s1   #pA2              ; Recognized by CASE.
       s1   #PA2              ; Recognized by CASE.
```

Duplication (DUP) 6.4

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

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

You can specify the DUP pseudo instruction anywhere within a program segment. If the DUP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the DUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the DUP pseudo instruction is as follows:

```
[dupname]       DUP       expression[,[count]]
```

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

- *dupname*

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

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

- *expression*

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

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

- *count*

  count is an optional absolute expression with positive value that specifies the number of statements to be duplicated. All symbols (if any) must be defined previously. If the current base is mixed, octal is used for the expression.

LOCAL pseudo instructions and comment statements (* in column 1) are ignored for the purpose of this count. Statements are counted before expansion of nested macro or opdef calls, and dup or echo sequences.

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

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

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

The following is the file, ALPHA:

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

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

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

The following example illustrates the expansion of the preceding example:

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

Duplicate with varying argument (ECHO)

6.5

The ECHO pseudo instruction defines a sequence of code that is assembled zero or more times immediately following the definition. On each repetition, the actual arguments are substituted for the formal parameters until the longest argument list is exhausted. Null strings are substituted for the formal parameters after shorter argument lists are exhausted. The echo sequence to be repeated consists of statements following the ECHO pseudo instruction and any optional LOCAL pseudo instructions. Comment statements are ignored. The echo sequence ends with an ENDDUP that has a matching location field name.

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

You can specify the ECHO pseudo instruction anywhere within a program segment. If the ECHO pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the ECHO pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the ECHO pseudo instruction is as follows:

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

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

- *dupname*

  dupname specifies the required name of the echo sequence. It must match the location field name in the ENDDUP instruction that terminates the echo sequence. *dupname* must meet the requirements for names as described in subsection 4.2, page 67.

- *name*

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

- *argument*

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

  A single argument is any ASCII character up to but not including the element separator, a space, a tab (new format only), or a semicolon (new format only). The first character cannot be a left parenthesis.

  A parenthesized list can be a list of one or more actual arguments. Each actual argument can be one of the following:

  – An ASCII character string can contain embedded arguments. If, however, an ASCII string is intended, the first character in the string cannot be a left parenthesis. A legal ASCII string is 4(5). An illegal ASCII string is (5)4(5).

  – A null argument; an empty ASCII character string.

  – An embedded argument that contains a list of arguments enclosed in matching parentheses. An embedded argument can contain blanks or commas and matched pairs of parentheses. The outermost parentheses are always stripped from an embedded argument when an echo definition is expanded.

    An embedded argument must meet the requirements for embedded arguments as described in subsection 4.7, page 94.

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

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

The following example illustrates the expansion of the preceding example:

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

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

```
         list             dup
example  echo             param1=,param2=()
                                ; ECHO with two null parameters.
_*Parameter 1 is:           'param1'
_*Parameter 2 is:           'param2'
example  enddup
```

The following illustrates the expansion of the preceding example:

```
*Parameter 1 is:       ''
*Parameter 2 is:       ''
```

Ending a macro or operation definition (ENDM)

6.6

An ENDM pseudo instruction terminates the body of a macro or opdef definition. If ENDM is used within a MACRO or OPDEF definition with a different name, it has no effect.

You can specify the ENDM pseudo instruction only within a macro or opdef definition. If the ENDM pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the ENDM pseudo instruction is as follows:

```
func      ENDM      ignored
```

The *func* variable associated with the ENDM pseudo instruction identifies the name of the macro or opdef definition sequence. It must be a valid identifier or the equal sign. *func* must match the functional that appears in the result field of the macro prototype or the location field name in an OPDEF instruction.

If the ENDM pseudo instruction is encountered within a definition but *func* does not match the name of an opdef or the functional of a macro, the ENDM instruction is defined and does not terminate the opdef or macro definition in which it is found. *func* must meet the requirements for functionals.

Premature exit from a macro expansion (EXITM)

6.7

The EXITM pseudo instruction immediately terminates the innermost nested macro or opdef expansion, if any, caused by either a macro or an opdef call. If files were included within this expansion and/or one or more dup or echo expansions are in progress within the innermost macro or opdef expansion they are also terminated immediately. If such an expansion does not exist, the EXITM pseudo instruction issues a caution level listing message and does nothing.

You can specify the EXITM pseudo instruction anywhere within a program segment. If the EXITM pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the EXITM pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the EXITM pseudo instruction is as follows:

```
ignored      EXITM      ignored
```

In the following example of a macro call, the macro expansion is terminated immediately by the EXITM pseudo instruction and the second comment is not included as part of the expansion:

```
       macro
       alpha
_*First comment
       exitm
_*Second comment
alpha  endm
       list   mac
```

The following example illustrates the expansion of the preceding example:

```
       alpha                    ; Macro call
*First comment
       exitm
```

Ending duplicated code (ENDDUP)

6.8

The ENDDUP pseudo instruction ends the definition of the code sequence to be repeated. An ENDDUP pseudo instruction terminates a dup or echo definition with the same name. If ENDDUP is used within a DUP or ECHO definition with a different location field name, it has no effect. ENDDUP has no effect on a dup definition terminated by a statement count; in this case, ENDDUP is counted.

The ENDDUP pseudo instruction is restricted to definitions (DUP or ECHO). If the ENDDUP pseudo instruction is found on a MACRO or OPDEF definition, it is defined and is not recognized as a pseudo instruction. If the ENDDUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the ENDDUP pseudo instruction is as follows:

dupname	ENDDUP	ignored

The *dupname* variable associated with the ENDDUP pseudo instruction specifies the required name of a dup sequence. *dupname* must meet the requirements for names as described in subsection 4.2, page 67.

Premature exit of the current iteration of duplication expansion (NEXTDUP)

6.9

The NEXTDUP pseudo instruction stops the current iteration of a duplication sequence indicated by a DUP or an ECHO pseudo instruction. Assembly of the current repetition of the dup sequence is terminated immediately and the next repetition, if any, is begun.

Assembly of the current iteration of the innermost duplication expansion with a matching location field name is terminated immediately. If the location field name is not present, assembly of the current iteration of the innermost duplication expansion is terminated immediately.

If other dup, echo, macro, or opdef expansions were included within the duplication expansion to be terminated, these expansions are also terminated immediately. If a file also is being included at expansion time within the duplication expansion it is terminated immediately.

You can specify the NEXTDUP pseudo instruction anywhere within a program segment. If the NEXTDUP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the NEXTDUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo.

The format of the NEXTDUP pseudo instruction is as follows:

```
[dupname]       NEXTDUP       ignored
```

The optional *dupname* variable specifies the name of a dup sequence. If the name is present but does not match any existing duplication expansion, a caution-level listing message is issued and the pseudo instruction does nothing. If the name is not present and a duplication expansion does not currently exist, a caution-level listing message is issued and the pseudo instruction does nothing.

Stopping duplication (STOPDUP)

6.10

The STOPDUP pseudo instruction stops duplication of a code sequence indicated by a DUP or ECHO pseudo instruction. STOPDUP overrides the repetition count.

Assembly of the current dup sequence is terminated immediately. STOPDUP terminates the innermost dup or echo sequence with the same name as found in the location field. If no location field name exists, STOPDUP will terminate the innermost dup or echo sequence. STOPDUP does not affect the definition of the code sequence that will be duplicated.

Assembly of the innermost duplication expansion with a matching location field name is terminated immediately; however, if the location field name is not present, assembly of the innermost duplication expansion is terminated immediately. If other dup, echo, macro, or opdef expansions were included within the duplication expansion that will be terminated, these expansions also are terminated immediately. If a file also is being included at expansion time within the duplication expansion that will be terminated, the inclusion of that file is terminated immediately.

You can specify the STOPDUP pseudo instruction anywhere within a program segment. If the STOPDUP pseudo instruction is found within a definition, it is defined and is not recognized as a pseudo instruction. If the STOPDUP pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the STOPDUP pseudo instruction is as follows:

```
[dupname]       STOPDUP       ignored
```

The *dupname* variable associated with the STOPDUP pseudo instruction specifies the name of a dup sequence. If the name is present but does not match any existing duplication expansion, or, if the name is not present and a duplication expansion does not currently exist, a caution-level listing message is issued and the pseudo instruction does nothing. *dupname* must meet the requirements for names as described in subsection 4.2, page 67.

The following example uses a DUP pseudo instruction to define an array with values 0, 1, and 2:

```
S     =     W*
      DUP   3,1
      CON   W.*-S
```

The following illustrates the expansion of the preceding example:

```
      CON   W.*-S
      CON   W.*-S
      CON   W.*-S
```

In the following example the ECHO and DUP pseudo instructions define a nested duplication:

```
ECHO   ECHO              RI=(A,S),RJK=(B,T)
I      SET               0
DUPI   DUP               8
JK     SET               0
DUPJK  DUP               64
       RI.I              RJK.JK
JK     SET               JK+1
DUPJK  ENDDUP
I      SET               I+1
DUPI   ENDDUP
ECHO   ENDDUP
```

Note: The following expansion is not generated by CAL, but it is included to show the expansion of the previously nested duplication expansion.

The following example illustrates the expansion of the preceding example:

```
                                ; In the first call of the echo, the A
                                ; and B parameters are used.
      A.0       B.0             ; DUPJK generates the A.O gets register
      .         .               ; B.O through register A.0 gets register
      .         .               ; B.64 instructions.
      .         .
      A.0       B.64            ; DUPI increments the A register from
      .         .               ; A.1 to A.7 for succeeding passes
      .         .               ; through DUPJK.
      .         .
      A.1       B.0             ; DUPJK generates register A.i gets
      .         .               ; register B.0 through register A.i gets
      .         .               ; register B.64 instructions.
      .         .
      A.7       B.64            ;  i is l to 7.
      S.0       T.O             ; In the second expansion of the echo
      .         .               ; pseudo instruction the S and T
      .         .               ; parameters are used.
      .         .
      S.0       T.64            ; DUPJK and DUPI generate the same
      .         .               ; series of register instructions for
      .         .               ; the S and T registers that were
      .         .               ; generated for the A and B registers.
      S.8       T.64
```

In the following example the STOPDUP pseudo instruction terminates duplication:

```
      LIST      DUP
T     SET       0
A     DUP       1000
T     SET       T+1
      IFE       T,EQ,3,1        ; Terminate duplication when T=3.
A     STOPDUP
      CON       T
A     ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
T       SET     T+1
        CON     T
T       SET     T+1
        CON     T
T       SET     T+1
A       STOPDUP
```

In the following example a STOPDUP pseudo instruction is used to terminate a DUP immediately:

```
DNAME   DUP             3
_* First comment
        STOPDUP
_* Second comment
DNAME   ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
* First comment
        STOPDUP
```

The following example is similar to the previous example except NEXTDUP replaces STOPDUP. The current iteration is terminated immediately when the NEXTDUP pseudo instruction is encountered.

```
DNAME   DUP             3
_* First comment
        NEXTDUP
_* Second comment
DNAME   ENDDUP
```

The following example illustrates the expansion of the preceding example:

```
* First comment
        NEXTDUP
* First comment
        NEXTDUP
* First comment
        NEXTDUP
```

Specifying local unique character string replacements (LOCAL)

6.11

The LOCAL pseudo instruction specifies unique character string replacements within a program segment that are defined only within the macro, opdef, dup, or echo definition. These character string replacements are known only in the macro, opdef, dup, or echo at expansion time. The most common usage of the LOCAL pseudo instruction is for defining symbols, but the LOCAL pseudo instruction is not restricted to the definition of symbols. Local pseudo instructions within a macro, opdef, dup, or echo header are not part of the macro definition.

On each macro or opdef call and each repetition of a dup or echo definition sequence, the assembler creates a unique 8-character string (commonly used for the definition of symbols by the user) for each local parameter and substitutes the created string for the local parameter on each occurrence within the definition. The unique character string created for local parameters has the form %%*nnnnnn*; where *n* is a decimal digit.

Zero or more LOCAL pseudo instructions can appear in the header of a macro, opdef, dup, or echo definition. The LOCAL pseudo instructions must immediately follow the macro or opdef prototype statement or DUP and ECHO pseudo instructions, except for intervening comment statements.

You can specify the LOCAL pseudo instruction only within a definition. If the LOCAL pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the LOCAL pseudo instruction is as follows:

```
ignored    LOCAL       [name][,[name]]
```

The *name* variable associated with the LOCAL pseudo instruction specifies formal parameters that must be unique and will be rendered local to the definition. *name* must meet the requirements for names as described in subsection 4.2, page 67.

The following example demonstrates that all formal parameters must be unique:

```
      MACRO
      UNIQUE   PARM2           ; PARM2 is defined within UNIQUE.
      LOCAL    PARM1,PARM2     ; ERROR: PARM2 previously defined as a
      .        .               ; parameter in the macro prototype
      .        .               ; statement.
      .        .
      UNIQUE   ENDM
```

The following example demonstrates how a unique character string is generated for each parameter defined by the LOCAL pseudo instruction:

```
        macro
        string
        local  param1,param2   ; Not part of the definition body.
param1  =      1
        s1     param1          ; Register s1 gets the value defined by
                               ; param1.
param2  =      2
        s2     param2          ; Register s2 gets the value defined by
                               ; param2.
string  endm                   ; End of macro definition.
        list   mac             ; Listing expansion.
```

The following example illustrates the call and expansion from the preceding example:

```
         string              ; Macro call.
%%262144 =                   1
         s1                  %%262144
                             ; Register s1 gets the value defined by
                             ; param1.
%%131072 =                   2
         s2                  %%131072
                             ; Register s2 gets the value defined by
                             ; param2.
```

The call to the macro string generates unique strings for `param1` (`%%262144`) and for `param2` (`%%131072`).

Synonymous operations (OPSYN)

6.12

The `OPSYN` pseudo instruction defines an operation that is synonymous with another macro or pseudo instruction operation. The functional name in the location field is defined as being the same as the functional name in the operand field. You can redefine any pseudo instruction or macro in this manner.

The functional name in the location field can be a currently defined macro or pseudo instruction in which case, the current definition is replaced and a message is issued informing you that a redefinition has occurred.

An operation defined by `OPSYN` is global if the `OPSYN` pseudo instruction occurs within the global part of an assembler segment, and it is local if the `OPSYN` pseudo instruction appears within an assembler module of a segment. You can reference global operations in any program segment following the definition. Every local operation is removed at the end of a program module, making any previous global definition with the same name available again.

If the `OPSYN` pseudo instruction occurs within a definition, it is defined and is not recognized as a pseudo instruction. If the `OPSYN` pseudo instruction is found within a skipping sequence, it is skipped and is not recognized as a pseudo instruction.

The format of the OPSYN pseudo instruction is as follows:

```
func1      OPSYN      [func2]
```

The *func1* variable associated with the OPSYN pseudo instruction specifies a required functional name. It must be a valid functional name. The name of a defined operation such as a pseudo instruction or macro, or the equal sign. *func*1 must not be blank and must meet the requirements for functional names.

The *func2* variable specifies an optional functional name. It must be the name of a defined operation or the equal sign. If *func*2 is blank, *func*1 becomes a do-nothing pseudo instruction.

In the following example, the macro definition includes the OPSYN pseudo instruction that redefines the IDENT pseudo instruction:

```
IDENTT   OPSYN           IDENT
         MLEVEL          ERROR ; Eliminates the warning error that is
                               ; issued because the IDENT pseudo
                               ; instruction is redefined.
         MACRO
         IDENT           NAME
         LIST            LIS,OFF,NXRF
NAME     LIST            LIS,ON,XRF
                               ; Processed if LIST=NAME on CAL control
                               ; statement.
         IDENTT          NAME
IDENT    ENDM
```

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

```
      IDENT    A
      LIST     LIS,OFF,NXRF
A     LIST     LIS,ON,XRF      ; Processed if LIST=NAME on CAL control
                               ; statement.
      IDENTT   A
```

In the following example, the `first` macro illustrates that a functional can be redefined many times:

```
         macro
         first
         s1             1
         s2             2
         s3             s1+2
first    endm
second   opsyn          first
                           ; second is the same as first.
third    opsyn          second
                           ; third is the same as second.
```

The following example includes the Opdef calls and expansions from the preceding example:

```
      first                    ; Macro call.
      s1       1
      s2       2
      s3       s1+s2
      second
      s1       1
      s2       2
      s3       s1+s2
      third
      s1       1
      s2       2
      s3       s1+s2
```

In the following example, the functional `EQU` is defined to perform the same operation as `=`:

```
EQU   OPSYN  =           ; EQU is defined to
                         ; perform the
                         ; operation that the
                         ; = pseudo
                         ; instruction
                         ; performs.
```