Chapter 7

dpas Lexical Directives

The **dpas** assembler includes an internal lexical preprocessor. Preprocessing is controlled via embedded lexical directives, very similar in functionality to the C language preprocessor, **cpp**. In fact, the **-P** switch to **dpas** allows the **cpp** preprocessor to be used instead of the internal mechanism. Moreover, the two can be mixed, so that advanced features of the internal preprocessor can be used along with the standard **cpp** preprocessing.

The **dpas** lexical directives provide lexical control over the DPEAC source, enhancing ease of hand coding. Lexical directives are lines inserted into the source text that begin with optional spaces followed by one of the following **dpas** directive keywords:

#comment	**#define**	**#else**	**#endif**
#endmacro	**#endrepeat**	**#ident**	**#if**
#ifz	**#ifdef**	**#ifndef**	**#ifsame**
#ifnsame	**#ifblank**	**#ifnblank**	**#include**
#macro	**#repeat**	**#set**	

Lexical directives are not passed to the output. Instead, they are interpreted during assembly to provide lexical control over the source stream. For example, directives can provide for

- Substitution of strings inside non-directive lines (**#define**)
- Conditionalization of source code (**#if**, etc.)
- An “include” file facility (**#include**)
- A fairly extensive macro processing capability (**#macro**)

As a simple example, consider the source text:

```
#define VECTOR_LENGTH 8
     ...
faddv*VECTOR_LENGTH V1,V2,V3
```

In the example, the lexical symbol **VECTOR_LENGTH** has been defined to be equivalent to the string “8”. In all future lines containing that lexical symbol, the string 8 will be substituted. Thus, the resulting DPEAC instruction from the example above becomes:

```
faddv*8 V1,V2,V3
```

The following sections discuss the syntactic elements of the lexical directives. You may notice that certain similarities and differences exist between the elements of the lexical directive sytax and those of DPEAC. Most differences result from an attempt to be compatible with the **cpp** preprocessor. A serious attempt to minimize such differences (through careful design of the DPEAC syntax) has been made, for example, in the constant and expression syntax.

7.1 Numeric Constants

Numbers are 64-bit constants. Parsing of constants follows the conventions of the C language.

Any item starting with a digit is assumed to be a constant. Any constant starting with a non-zero digit, and consisting only of digits is assumed to be a decimal (radix 10) number, e.g. ”1234”, ”1985”.

Any number starting with zero, and consisting entirely of digits, is assumed to be an octal (radix 8) number. Alternate radices can be specified using radix prefixes, as follows:

`0x1AF`...,	`0X1AF`...	Hexidecimal constants
`0b101`...,	`0B101`...	Binary Constants
`0o123`...,	`0O123`...	Octal constants
`0n987`...,	`0N987`...	Decimal constants

Floating point constants in various forms may also be represented.

Floating point numbers are designated as follows:

`0`*t*`[+,-]`*nnn*`[.[`*nnn*`][{E,e}[+,-]`*nnn*`]`

where *t* indicates the type, one of:

`f`	(or `F`)	IEEE Single Precision (32 bit)
`r`	(or `R`)	IEEE Single Precision (32 bit)
`d`	(or `D`)	IEEE Double Precision (64 bit)

For example:

```
0f0.0              0f-1.0          0d6.02E23
0F.31415E+1        0D.000001       0f1
```

Constants prefixed by “`0f`” are IEEE single precision constants having the form:

s	*exponent*	*fraction*
1	8	23

where *s* is the sign (1 is negative), *exponent* is an excess-127 binary exponent, and *fraction* is the 24-bit binary characteristic. (The 24th bit of the fraction is ”hidden” and is assumed to be one, i.e. the constant is normalized.)

DPEAC also provides the alternate “`0r`” single-precision syntax for compatibility with the SPARC assembler. This format does not have special syntax for double precision constants.

Constants prefixed by “`0d`” are IEEE double precision floating point constants, having the form:

s	*exponent*	*fraction*
1	11	52

where fields are as before, except that the *exponent* is excess-1023 and 32 additional bits of significance are added to the *fraction* part.

NOTE

Although the IEEE double precision format allows exponents in the range –1023 to +1023, **`dpas`** running on a VAX currently restricts such constants to exponents in the range –127 to +127. Constants outside of this range must be hand-encoded and given in hex as integers.

Constants composed of ASCII characters may also be specified by enclosing them in apostrophes, as follows:

`'A'`	is equivalent to	`0x41`, `0101`, `65` and `0b1000001`
`'ABC'`	is equivalent to	`0x414243`

The second example shows that '**`ABC`**' is not equivalent to the string "**`ABC`**", in that characters are packed in arithmetic (radix 256) order, rather than in byte storage order.

7.2 Expressions

An expression can be inserted in any place where a numerical quantity is called for in the syntax. The expression syntax is identical to the *constant-expression* syntax in the DPEAC language, with the addition of the "**`defined`**" operator, described below.

Expressions are numbers, symbols, and operators. When a symbol is used, it must have a numeric value, or a text value which can be interpreted (using the rules of number syntax) as a numeric value. All values are 64-bit integers and are computed using 64-bit operations.

The table below lists the operators that may be used, from highest to lowest evaluation precedence. (Note, though, that operator precedence can be overridden with the use of parentheses.)

Unary Operators:

`+` Plus (unary, has no effect)
`-` Negate (2's complement)
`!` Logical not
`~` Bitwise 1's Complement (invert)
`%lo` Extract low 10 bits (and with `0x3FF`)
`%hi` Extract high 22 bits (shift right 10)

Symbol Definition Predicates:

`defined(`*symbol*`)` The value 1 if *symbol* is defined, else 0
`defined` *symbol* Same as defined(symbol)

Bitwise Logical Operators: (Evaluated Left To Right)

`&` Bitwise AND
`|` Bitwise OR
`^` Bitwise XOR

Scaling Operators: (Evaluated From Left To Right)

`*` Signed multiply
`/` Signed divide
`<<` Logical Left shift (as in "C")
`>>` Logical Right shift (as in "C")

Offsetting Operators: (Evaluated From Left To Right)

`+` Addition
`-` Subtraction

Comparison Operator: (Yield Value Of 1=True, Or 0=False)

`<` Less than (unsigned)
`<=` Less than or equal to (unsigned)
`==` Equal to
`!=` Not equal to (`<>` is also allowed)
`>` Greater than (unsigned)
`>=` Greater than or equal to (unsigned)

Logical Operators: (Yield Values 0 Or 1 Only)

`&&` Logical And
`||` Logical OR

The use of floating point constants in expressions may be misleading. Recall that expression computation is all 64-bit integer calculation. Floating point number in expressions are converted to their 32 or 64-bit bit pattern (e.g. 0f1.0 becomes #x3f000000) before being used in computation. Single-precision constants convert to the low 32 bits of the 64 bit resulting value.

It is important not to confuse **dpas** lexical expression syntax with that expected for *as-expressions* (shown in the DPEAC syntax descriptions). *as-expressions* are passed to the output for processing by the IU assembler, while lexical expressions are evaluated during the processing of **dpas** lexical syntax.

7.3 String Constants

Some places in the **dpas** syntax allow ASCII strings to be given. Strings are enclosed in quotation marks (“”).

To supply a literal quotation mark, double it. For example, the string:

```
"My name is ""Sam"", what's yours?"
```

specifies the literal characters:

```
My name is "Sam", what's yours?
```

Where used to specify numeric data, a string represents the sequence of bytes containing the quoted characters left-to-right in storage order.

7.4 Symbols, #define, #set, and #undef

dpas maintains its own symbol table. Since **dpas** is used as a preprocessor for assembly code, it is important to realize that the **dpas** symbol table is distinct from that of the assembler itself. From here on, we shall refer to **dpas** symbols simply as “symbols”.

Symbols are constructed using upper and lower case letters, dollar signs (“**$**”) and underscores (“**_**”). When **dpas** is operated in the case-senstive mode (the default), symbols which differ only in the case in which they are set are distinct.

7.4.1 #define

Symbols are defined and bound to a value using the **#define** directive. The value of a symbol is a string of text.

The syntax of the **#define** directive is:

#define *symbol text*...

text is any sequence of characters, including spaces. Spaces and tabs before and after text are discarded, as are all characters including and after a comment character ("**!**"). For example:

```
#define NAME OTHER_NAME    ! comment
#define length 7
#define extent length * 4
```

Note that in the last example, the symbol **extent** is not defined as 28, but rather as the text "**length * 4**".

A parameterized version of **#define** is also available, and is documented below under "Parameterized **#define**".

7.4.2 #set

A variant on the **#define** directive is the **#set** directive. **#set** also defines a symbol, but takes an expression (in the lexical syntax, described below) as its argument. It evaluates that expression, and sets the symbol to the resulting value expressed as a signed decimal number. This is useful for forcing early expression evaluation, or for avoiding lexical repetition of long expressions, which can slow down **dpas**. The syntax of the **#set** directive is:

#set *symbol expression*

where *expression* is a legal numerical expression (as described in the next section). It is important to realize that symbol substitution is performed on expression prior to evaluation. Thus *expression* can include symbols and values defined by and in the syntax of the lexical directives. Symbol substitution is not performed on the *symbol* being set.

For example, given the code

```
#set length 7
#set extent length * 4
```

dpas substitutes “**7**” for “**length**”. It then evaluates 7*4 and defines **extent** as the string “28”.

7.4.3 #undef

It is possible to remove the definition of a symbol using the **#undef** directive. The syntax is:

#undef *symbol*

An undefined symbol will no longer be substituted in subsequent text. In all ways, it will be as if the symbol had never been defined.

7.4.4 Predefined Symbols

Before lexical processing begins, **dpas** pre-defines the symbol “**dpas**” to the string “1”. This allows source programs to test whether they are being assembled by **dpas** or not. This definition is also done if the **cpp** preprocessor is used, by passing the switch “**–Ddpas=1**” to cpp.

7.5 Symbol Substitution

Once a symbol is defined, it will be replaced in subsequent source code with its defined value. This substitution is done first, before any other lexical processing, including macro definition or expansion. **dpas** performs symbol substitution immediately after reading the source line. Substitution is done recursively until no defined symbols appear in the source line, after which the line is parsed. Infinitely recursive substitution is detected and results in an error.

No symbol substitution is performed:

1. Anywhere within a **`#define`** directive.
2. On the target symbol in a **`#set`** directive (i.e. the symbol being set). However, substitution is performed on the expression portion prior to evaluation.
3. Within **`#ifdef`** or **`#ifndef`** directives.
4. During the definition of a macro, i.e. within and anywhere between **`#macro`** and **`#endmacro`** directives.
5. During the parsing of the actual parameters in a macro invocation.

7.6 #comment

This directive is ignored by **`dpas`** and is not passed to the output. Syntax is:

`#comment` *text ...*

7.7 #include

The **`#include`** line causes the named file to be included in the input stream. The format of the **`#include`** lexical line is one of:

`#include` *filename*
`#include` **"***filename***"**
`#include` **<***filename***>**

If the unquoted syntax is used, the include file is opened using the specified filename, i.e. in the current working directory.

In any of the three cases, if the include file begins with a "**/**" character, the file is looked up using that given path and name only.

If the quoted syntax is used (and the filename does not begin with "**/**"), an internal include file search list is used to find the specified file. This is done by traversing the search list and appending each list element (a directory path) to the specified file name (with a "**/**" in between) and attempting to open the file with that name. The search list consists of the following directories, in this order:

1. The directory containing the currently active source file (in which the **#include** was encountered).

2. The list of directories specified as arguments to one or more **-I** switches to **dpas** (in order).

3. **/usr/include/**.

   NOTE: The angle-bracketed syntax functions identically to the quoted syntax, except that the directory containing the the current source file is not searched.

   The quoted and angle-bracketed syntax are identical in function to the **cpp** preprocessor syntax. The unquoted syntax is not compatible with **cpp**.

Once found, the included file is opened and subsequent input lines are taken from it. When the end of the included file is reached, it is closed, and the inputting of source lines following the **#include** line is resumed. **#include** files may be nested up to 10 deep.

7.8 Conditional Assembly (#if, #else, and #endif)

The **#if***xxx* lexical statements allow subsequent text to be conditionally passed to the output of **dpas**. Thus, they allow a block of code to be either included or ignored, based on the testing of a specific condition. Testing can include the evaluation of an expression, the equality of two strings, and so on.

Optionally, an **#else** line can be included folllowing the body of code being conditionalized. The **#else** marks the beginning of another body of code that is processed or ignored depending on the *inverse* evaluation of the **#if** expression.

The **#endif** lexical statement ends the conditional body of code, and *is required*.

It should be noted that **#repeat** blocks (see below) within deselected code, are expanded, even though the resulting lines are not processed. Macro definitions within deselected code however, are not processed.

7.8.1 #if and #ifz

The form of the **#if** and **#ifz** statements is:

#if[**z**] *expression*

When encountered, the given *expression* is evaluated. It must be a “pass1–time” expression which evaluates to an absolute value. For **#if**, the lines following will be processed, unless the specified *expression* evaluates to zero. **#ifz** is the opposite, it enables subsequent text only if the *expression* evaluates to exactly zero. Failure to evaluate the *expression* results in an error diagnostic, with the *expression* substituted with a value of zero.

For example:

```
#set DOIT 1
#if DOIT
  xxxxxxxx           !This line will be processed
#else
  yyyyyyyy           !This line will NOT be processed
#endif
```

7.8.2 #ifdef, #ifndef

These lexical functions operate like **#if**, except that they test whether a named symbol has been defined or not. The format is:

#ifdef *symbol*
#ifndef *symbol*

The **#ifdef** statement allows code up to the matching **#else** or **#endif** statement to be evaluated if the named *symbol* is defined. The *symbol* may be the name of a macro, a structure, an arbitrary symbol defined by **#define**, etc., as long as it has a definition when the **#if**[**n**]**def** statement is encountered.

The **#ifndef** function works as the inverse of the **#ifdef** function, enabling subsequent code only if the named symbol has *not* been defined.

Note that these statements do not evaluate the *symbol* argument — they treat it literally; it is not replaced by its **#define** or **#set** definition, if any.

7.8.3 #ifsame, #ifnsame

The **#ifsame** and **#ifnsame** statements are also like **#if**, and control conditionalization based on whether or not two character strings are equal or different. The format is:

#if[**n**]**same** "*string1*", "*string2*"

For example, the following statement enables the subsequent code:

```
#ifsame  "hello","hello"
```

but the following two examples both disables subsequent code

```
#ifsame  "hello","goodbye"
#ifsame  "hello","HELLO"
```

#ifnsame works in the inverse fashion, causing subsequent lines not to be processed if the strings match.

7.8.4 #fblank, #ifnblank

The **#ifblank** and **#ifnblank** statements control conditionalization based on whether or not the remainder of the line is blank or not. This is typically used to test if a macro parameter is null (blank). The format is:

#if[**n**]**blank** *text*

For example, the following statement enables subsequent code:

```
#define  BLANK
#ifblank BLANK
```

but the following disables subsequent code:

```
#define  NONBLANK  %$RE*((
#ifblank NONBLANK
```

#ifnblank works in the inverse fashion, causing subsequent lines *not* to be processed if there is no text following in the line.

7.8.5 The #else if Construction

For the purpose of constructing "case" style tests, the **#else** statement may be extended to include a subsequent **#if**. This is identical to the **#elif** statement in ANSI C, and the **else-if** construct available in some programming languages.

For example, the following code replaces the value of the the symbol **x** with **x^x** as long as it is 1, 2, or 3. Otherwise, it prints an error message:

```
#if x==1
    #set x  x
#else if x==2
    #set x  x*x
#else if x==3
    #set x  x*x*x
#else
    #error "Illegal value for x"
#endif
```

All legal **#if**xxx forms may be used following the **#else**, i.e. **#ifsame**, **#ifblank**, etc.

7.9 Parameterized #define

In its simplest form, the **#define** statement defines a direct substitution of an expression for a given symbol. However, you can also supply "arguments" to a **#define** macro to affect the resulting substituted expression.

For example, let us define a symbol "**SA**" as shown in the second line below:

```
#define STACK_ALIGN 8
#define SA(X) (((X)+(STACK_ALIGN-1)) & ~(STACK_ALIGN-1))
```

The parameter "**X**", is called a *formal parameter*. There must be no spaces between the symbol name and the open parenthesis of the parameter list "(X)", or else the list will be considered part of the expression to be substituted. Note that in the parameterized form of **#define**, the text of the definition is *not* subjected to symbol substitution; hence the symbol **STACK_ALIGN** is not substituted in the definition itself. It is only substituted when the **#define**-d symbol is applied.

All further uses of the symbol **SA** must be accompanied by an *actual parameter* to be substituted for **X** in the replacement string. To complete the example, assume that some later line contains the text:

```
add     %i2,SA(9),%i2
```

Here, the text "SA(9)" would be replaced by the string:

```
(((9)+(STACK_ALIGN-1)) & ~(STACK_ALIGN-1))
```

as a result of replacing all instances of the formal parameter "**X**" (just one here) with the actual parameter "9". Since symbol substitution is done recursively, the symbol "**STACK_ALIGN**" is then replaced with its simple defined value of "8". The resulting line is finally:

```
add     %i2,(((9)+(8-1)) & ~(8-1)),%i2
```

Source Note: the **SA** example above is taken from actual source code in the header file **/usr/include/sparc/asm_linkage.h**. **SA** aligns a value up to the next multiple of 8, i.e. aligns a value to a doubleword boundary.

Up to 32 formal parameters may be specified. All subsequent uses of the defined symbol must bear the exactly correct number of matching actual parameters.

Incompatibility Note: Readers familiar with the **cpp** preprocessor will note that the rules of definition and substitution are the same, with one small exception. **cpp** allows the actual parameters to span multiple lines, whereas **dpas** does not.

```
#define plus(a,b)  a + b
! this is not legal:
plus
  (a,
   b)
```

7.10 Multiple-line Macros (#macro and #endmacro)

Macros give the programmer a way to generate code from a template, and to substitute parameters into the template.

Use of a macro has two stages: definition and expansion.

- Definition occurs when the code that forms the macro is assembled. During definition, the following actions occur:
  - The macro name becomes known and associated with the macro.
  - The formal parameters for the macro are established; each gets a symbol name and, optionally, a default value.
  - Optionally, further formal parameters are defined as parameter markers, within the body of the macro
  - Lexical lines (**`#define`**, **`#if`**, etc.) are processed, unless they have been prefaced by an extra # (**`##define`**, **`##if`**, etc.).

Once a macro has been defined, it remains defined until it is redefined, until it is undefined with an **`#undef`** instruction, or until the program terminates. Macros may be redefined freely during **`dpas`** processing.

- Once the macro is defined, it can be invoked by later code. When invoked, the macro is expanded:
  - Formal parameters are replaced by actual parameter values.
  - Parameter markers are likewise replaced by actual values.
  - Any lexical lines remaining in the macro are processed.

7.10.1 Definition

The **`#macro`** line specifies that the following body of lines, up to the next **`#endmacro`** line, are the body of a macro definition. The format of the **`#macro`** line is:

`#macro` *name formal–parameter*`[`**`=`***default*`], ...`
macro body
`...`
`#endmacro`

Each *formal-parameter* is a symbol. All formal parameters must be different from each other, but may be the same as any defined symbols, or as formal parameters from other macro definitions.

Each *formal-parameter* can optionally be assigned a *default*, which is the value assigned to the formal parameter during expansions in which the associated actual parameter is not given.

Parameter Markers

The body of the macro can include parameter markers of the form

{*formal-parameter***}**

Actual parameters, specified when the macro is invoked, will be substituted for these markers.

Note: If the symbol inside the braces has been defined using the **`#define`** directive, it is subject to substitution.

Predefined Markers

Several predefined parameter markers exist.

- The marker {**`_num_actuals`**} will be replaced with the decimal number of actual parameters that have been supplied for the current macro invocation.
- The marker {**`#n`**} will be replaced with the Nth actual parameter.

  Numbering starts with 1, not with 0. For example, the fifth actual parameter will be substituted for the marker {5}. If n is 0, or is greater than the number of actual parameters provided, the NULL string replaces the marker.
- The marker {**`*`**} will be replaced with a decimal number unique to the macro currently being expanded. Every time a macro is invoked, it is assigned a unique expansion index. Thus, use of {**`*`**} within a macro will result in a unique string which is constant throughout the expansion, but different from that used in all other expansions of the macro (or any other macro). As an example, consider the macro defined as:

```
#MACRO  test
         dpjump L{*}
    L{*}:inc %i3
#endmacro
```

The first time the macro is expanded, some unique value will be substituted for {*}, for example "123":

```
          dpjump L123
L123:     inc %i3
```

If the macro is expanded again, another unique number is assigned:

```
          dpjump L456
L456:     inc %i3
```

Usage Note: Expansion indices are not guaranteed to be assigned sequentially, merely to be unique.

Special Characters within Macros

A literal "**{**" may be specified by doubling, i.e. specifying "**{{**". Similarly, "**}}**" may be given to represent a literal "**}**".

A literal "**#**" may also be specified by doubling, i.e., specifying "**##**". This allows you to include lexical lines (such as **#if**, **#repeat**, **#include**, etc.) in the macro body for evaluation when the macro is expanded (by specifying **##if**, etc.). If you do not double the "**#**", then the lexical lines are processed during macro definition. Doubled "**##**" are converted to single "**#**" during definition.

Lexical Directive Evaluation

If a lexical directive (such as a #define or #set) is included in the body of a macro, it is evaluated *once*, when the macro is defined, and *not* when the macro is executed. As an example, consider the following definition:

```
#set i 0

#macro MAC_ONE
    #set i i+1
    .word i
#endmacro
```

In macro **MAC_ONE**, the "**#set i i+1**" directive is evaluated when the macro is defined. Thus, the symbol i has the value 2 during and after the definition of the macro. The body of this macro is the single line "**.word 2**".

Now consider:

```
#macro MAC_TWO
    ##set i i+1
    .word i
#endmacro
```

In macro **MAC_TWO**, the "**#set**" is not performed during definition, but is included into the macro body with the "**##**" converted to "**#**" Thus the body of this macro is

```
#set i i+1
.word i
```

Thus, each time **MAC_TWO** is expanded, it will expand into code that increments the value of **i** and generates a word containing the new value.

More About Macros

Macros may be freely redefined during **dpas** processing.

During the definition of a macro, i.e. between **#macro** and **#endmacro**, no symbol substitution from **#define** and **#set** definitions is performed. Thus, in the above example, the symbol **i** is not substituted into the macro definition.

Macro invocations may be nested arbitrarily deep. (That is, a macro can include, as part of its body, an invocation of the same macro or another macro.)

7.10.2 Expansion

Macros are invoked by issuing the macro name at the beginning of a line (possibly preceded by a label), along with a list of actual parameters, separated by commas, and delimited by the commas or spaces. For example:

```
my-macro argument1, argument2, argument3
```

If an actual parameter must have commas or spaces in it, it may be given as a string delimited by quotes. Literal quotes may be included in the string by doubling them (""). The actual parameters are substituted in order for the corresponding formal parameters, where marked in the definition, as the macro is expanded.

Actual parameters may be null, as in:

macroname `,,`*arg3*

in which case the default value given in the macro definition will be used, or null (empty) text if no default was given. Missing trailing actuals are treated in the same fashion, even if sufficient commas are not given.

Usage Note: It is legal to supply more actuals than there are formals in the definition, but those actuals can only be expanded using the {**`#nnn`**} marker syntax, described above.

Positional Parameters and Keyword Parameters

The one-to-one association of actual parameters to formal parameters is called *positional* parameter specification. Parameters can also be given using *keyword* parameter specification, wherein the parameter to a macro is specified as follows:

formal–parameter = *actual–parameter*

In this type of specification, *formal–parameter* is one of the formal parameter names given in the macro definition, and *actual–parameter* is the actual parameter being specified. The actual parameter text may be quoted, etc. Spaces and tabs may be placed around the equal sign.

Mixing positional parameters and keyword parameters is allowed, but can be confusing. **`dpas`** will always bind positional parameters to formal parameters ordinally. Thus, the *N*th positional parameter is always bound to the *N*th formal parmater regardless of intervening keyword parameters.

Furthermore, **`dpas`** always processes actual parameters from left to right. As a result it is highly recommended that positional parameters be given first, followed by keyword parameters so that the one-for-one binding of positional parameters is clear in the code.

`dpas` will try to be as smart as possible when processing actual parameter values that contain equal signs. If an actual parameter specification is in the form "*symbol*`=`*text*" but *symbol* is not one of the formal parameter names, the entirety will be considered a positional parameter specification, equal sign and all.

7.10.3 Examples of Macro Substitution

As an example, consider the **threewords** macro defined below:

```
#macro threewords a,b,c
    .word {a},{b},{c}
#endmacro
```

The following table shows the expansions for various invocations of the macro threewords

Invocation	Expansion
`threewords 1,2,3`	`.word 1,2,3`
`threewords 1,2`	`.word 1,2,`
`threewords a=5,b=6,c=7`	`.word 5,6,7`
`threewords b=3`	`.word ,3,`
`threewords 1,2,a=3`	`.word 3,2,`
`threewords a=3,1,2`	`.word 1,2,`
`threewords a=10,b=20,30`	`.word 30,20,`
`threewords 1,2,d=20`	`.word 1,2,d=20`

7.10.4 Examples of Lexical Substitution

The **#ifsame** and **#ifnsame** lexical functions are typically used to compare #macro actual parameters to specific literal strings. **#ifblank** and **#ifnblank** are used to determine if an actual parameter is null (blank):

```
#macro   test   parm1,parm2
    ##ifsame "{parm1}","DELETE"   ! is parm1 = DELETE ?
    ##ifblank {parm2}             ! is parm2 null ?
#endmacro
```

7.11 #repeat, #endrepeat

The **`#repeat`** line specifies that the following body of code, up to the next **`#endrepeat`** line is to be processed *N* times in succession. The format is:

`#repeat` *repeat–count*
text to be repeated
`...`
`#endrepeat`

The *repeat–count* is an expression that evaluates to an integer quantity. Expressions not meeting this requirement are flagged as errors, and a *repeat-count* of one (1) is assumed.

`Usage Note:` This construct is identical to actual repetition of the given text.

When using **`#repeat`** within a macro invocation, note that the body of the **`#repeat`** is fully expanded, ie. all markers are substituted within the **`#repeat`** body, and *then* the text is repeated.

7.12 #print, #warning, and #error

These three directives print text to **`stderr`**. The syntax of these statements is:

`#print` *item*`,` *item*`, ...`
`#warning` *item*`,` *item*`, ...`
`#error` *item*`,` *item*`, ...`

where item is quoted ASCII text or an expression with optional format control.

When encountered, the listed items are printed to **`stderr`**. Each item may be either a quoted ASCII string, or a *constant-expression*.

For expressions, symbol substitution is first performed and then the expressions are evaluated and printed, by default as unsigned 64-bit hexidecimal numbers.

Following the expression, may be a format specifier of the form:

`:`*wwwf*

where *www* is a field width (a decimal constant).

The *f* component is a format specifier taken from the following table:

x,X Print as hexidecimal integer, padded with zeros
O,o Print as octal integer, padded with zeros
B,b Print as binary integer, padded with zeros
U,u Print as Decimal integer, padded with spaces
F,f Print as FLOAT, padded with spaces
D,D Print as DOUBLE, padded with spaces
A,a Print as ASCII characters, padded with spaces

If the printed value exceeds the requested field width, the value will be printed in full anyway. The field width, if omitted as in "**:x**", is taken as 1. The format, if omitted as in "**:10**", is taken as "**x**". If no format specifier is given, "**:1x**" is assumed.

ASCII strings, enclosed in quotes, are printed as given. Double quotation marks ("") may be used within the string to cause a single quotation mark to be printed.

Command line switches (**-p** and **-P**) are provided for supressing or redirecting **#print** output to a file.

The **#error** directive works exactly like **#print**, except that a processing error is signalled. This causes an error message to be printed to the user's terminal exactly as if a true assembly error had occurred. In addition, the output file will not be generated, and **dpas** will exit (later) with a non-zero exit status. **#error** output may not be suppressed or redirected.

The **#warning** directive also works exactly like **#print**, except that an assembly warning is signalled. This causes a warning message to be printed to the user's terminal exactly as if a true assembly warning had occurred, but does not otherwise affect procesing. **#warning** output may not be suppressed or redirected.

7.13 #ident

This little known, undocumented feature of the **cpp** preprocessor is supported by **dpas**. If encountered, the entire line is passed to the output unchanged. The as assembler will ignore it. The format is:

#ident *text to be passed directly to output*