m4 macro processor overview

This topic provides information about the **m4** macro processor, which is a front-end processor for any programming language being used in the operating system environment.

At the beginning of a program, you can define a symbolic name or symbolic constant as a particular string of characters. You can then use the **m4** macro processor to replace unquoted occurrences of the symbolic name with the corresponding string. Besides replacing one string of text with another, the **m4** macro processor provides the following features:

- Arithmetic capabilities
- File manipulation
- Conditional macro expansion
- String and substring functions

The **m4** macro processor processes strings of letters and digits called *tokens*. The **m4** macro processor reads each alphanumeric token and determines if it is the name of a macro. The program then replaces the name of the macro with its defining text, and pushes the resulting string back onto the input to be rescanned. You can call macros with arguments, in which case the arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

The **m4** macro processor provides built-in macros such as **define**. You can also create new macros. Built-in and user-defined macros work the same way.

Using the m4 macro processor

To use the **m4** macro processor, enter the following command:

```
m4 [file]
```

The **m4** macro processor processes each argument in order. If there are no arguments or if an argument is **-** (dash), **m4** macro processor reads standard input as its input file. The **m4** macro processor writes its results to standard output. Therefore, to redirect the output to a file for later use, use a command such as the following:

```
m4 [file] >outputfile
```

Creating a user-defined macro

Macro	Description
define (*MacroName***,** *Replacement***)**	Defines new macro *MacroName* with a value of *Replacement*.

For example, if the following statement is in a program:

```
define(name, stuff)
```

The **m4** macro processor defines the string `name` as `stuff.` When the string `name` occurs in a program file, the **m4** macro processor replaces it with the string `stuff.` The string `name` must be ASCII alphanumeric and must begin with a letter or underscore. The string `stuff` is any text, but if the text contains parentheses, the number of open, or left, parentheses must equal the number of closed, or right, parentheses. Use the **/** (slash) character to spread the text for `stuff` over multiple lines.

The open (left) parenthesis must immediately follow the word **define**. For example:

```
define(N, 100)
```

```
 . . .
if (i > N)
```

defines N to be 100 and uses the symbolic constant N in a later **if** statement.

Macro calls in a program have the following form:

```
name(arg1,arg2, . . . argn)
```

A macro name is recognized only if it is surrounded by non-alphanumeric characters. In the following example, the variable NNN is not related to the defined macro N.

```
define(N, 100)
 . . .
if (NNN > 100)
```

You can define macros in terms of other names. For example:

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100. If you later change the definition of N and assign it a new value, M retains the value of 100, not N.

The **m4** macro processor expands macro names into their defining text as soon as possible. The string N is replaced by 100. Then the string M is also replaced by 100. The overall result is the same as using the following input initially.

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N, so when the value of M is requested later, the result is the value of N at that time (because the M is replaced by N, which is replaced by 100).

Using the quote characters

To delay the expansion of the arguments of **define**, enclose them in quote characters. If you do not change them, quote characters are ` and ' (left and right single quotes). Any text surrounded by quote characters is not expanded immediately, but quote characters are removed. The value of a quoted string is the string with the quote characters removed. If the input is:

```
define(N, 100)
define(M, `N')
```

The quote characters around the N are removed as the argument is being collected. The result of using quote characters is to define M as the string N, not 100. The general rule is that the **m4** macro processor always strips off one level of quote characters whenever it evaluates something. This is true even outside of macros. To make the word define appear in the output, enter the word in quote characters, as follows:

```
`define' = 1;
```

Another example of using quote characters is redefining N. To redefine N, delay the evaluation by putting N in quote characters. For example:

```
define(N, 100)
. . .
define(`N', 200)
```

To prevent problems from occurring, quote the first argument of a macro. For example, the following fragment does not

redefine N:

```
define(N, 100)
. . .
define(N, 200)
```

The N in the second definition is replaced by 100. The result is the same as the following statement:

```
define(100, 200)
```

The **m4** macro processor ignores this statement because it can define only names, not numbers.

Changing the quote characters

Quote characters are normally ` and ' (left or right single quotes). If those characters are not convenient, change the quote characters with the following built-in macro:

Macro	Description
changequote (*l***,** *r* **)**	Changes the left and right quote characters to the characters represented by the *l* and *r* variables.

To restore the original quote characters, use **changequote** without arguments as follows:

```
changequote
```

Arguments

The simplest form of macro processing is replacing one string by another (fixed) string. However, macros can also have arguments, so that you can use the macro in different places with different results. To indicate where an argument is to be used within the replacement text for a macro (the second argument of its definition), use the symbol **$***n* to indicate the *n*th argument. When the macro is used, the **m4** macro processor replaces the symbol with the value of the indicated argument. For example, the following symbol:

```
$2
```

Refers to the second argument of a macro. Therefore, if you define a macro called bump as:

```
define(bump, $1 = $1 + 1)
```

The **m4** macro processor generates code to increment the first argument by 1. The bump(x) statement is equivalent to x = x + 1.

A macro can have as many arguments as needed. However, you can access only nine arguments using the **$***n* symbol ($1 through $9). To access arguments past the ninth argument, use the **shift** macro.

Macro	Description
shift (*ParameterList***)**	Returns all but the first element of *ParameterList* to perform a destructive left shift of the list.

This macro drops the first argument and reassigns the remaining arguments to the **$***n* symbols (second argument to $1, third argument to $2. . . tenth argument to $9). Using the **shift** macro more than once allows access to all arguments used with the macro.

The **$0** macro returns the name of the macro. Arguments that are not supplied are replaced by null strings, so that you can define a macro that concatenates its arguments as follows:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus:

```
cat(x, y, z)
```

is the same as:

```
xyz
```

Arguments $4 through $9 in this example are null because corresponding arguments were not provided.

The **m4** macro processor discards leading unquoted blanks, tabs, or new-line characters in arguments, but keeps all other white space. Thus:

```
define(a, b c)
```

defines a to be b c.

Arguments are separated by commas. Use parentheses to enclose arguments containing commas, so that the comma does not end the argument. For example:

```
define(a, (b,c))
```

has only two arguments. The first argument is a, and the second is (b,c). To use a comma or single parenthesis, enclose it in quote characters.

Using a predefined m4 macro

The **m4** macro processor provides a set of predefined macros. This section explains many of the macros and their uses.

Removing a macro definition

Macro	Description
undefine (`*MacroName*'**)**	Removes the definition of a user-defined or built-in macro (`*MacroName*')

For example:

```
undefine(`N')
```

removes the definition of N. After you remove a built-in macro with the **undefine** macro, as follows:

```
undefine(`define')
```

You cannot use the definition of the built-in macro again.

Single quotes are required in this case to prevent substitution.

Checking for a defined macro

Macro	Description
ifdef (`*MacroName*'**,** *Argument1***,** *Argument2***)**	If macro *MacroName* is defined and is not defined to zero, returns the value of *Argument1*. Otherwise, it returns *Argument2*.

The **ifdef** macro permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument. If the first argument is not defined, the value of **ifdef** is the third argument. If there is no third argument, the value of **ifdef** is null.

Using integer arithmetic

The **m4** macro processor provides the following built-in functions for doing arithmetic on integers only:

Macro	Description
incr (*Number***)**	Returns the value of *Number* + 1.

Macro	Description
decr (*Number***)**	Returns the value of *Number* - 1.
eval	Evaluates an arithmetic expression.

Thus, to define a variable as one more than the *Number* value, use the following:

```
define(Number, 100)
define(Number1, `incr(Number)')
```

This defines `Number1` as one more than the current value of `Number`.

The **eval** function can evaluate expressions containing the following operators (listed in descending order of precedence) :

unary **+** and **-**

****** or **^** (exponentiation)

*** / %** (modulus)

+ -

== != < <= > >=

!(not)

& or **&&** (logical AND)

| or **||** (logical OR)

Use parentheses to group operations where needed. All operands of an expression must be numeric. The numeric value of a true relation (for example, 1 > 0) is 1, and false is 0. The precision of the **eval** function is 32 bits.

For example, define `M` to be `2==N+1` using the **eval** function as follows:

```
define(N, 3)
define(M, `eval(2==N+1)')
```

Unless the text is very simple, use quote characters around the text that defines a macro

Manipulating files

To merge a new file in the input, use the built-in **include** function.

Macro	Description
include (*File***)**	Returns the contents of the file *File*.

For example:

```
include(FileName)
```

inserts the contents of `FileName` in place of the **include** command.

A fatal error occurs if the file named in the **include** macro cannot be accessed. To avoid a fatal error, use the alternate form, the **sinclude** macro (silent include).

Macro	Description
sinclude (*File* **)**	Returns the contents of the file *File*, but does not report an error if it cannot access *File*.

The **sinclude** (silent include) macro does not write a message, but continues if the file named cannot be accessed.

Redirecting output

The output of the **m4** macro processor can be redirected again to temporary files during processing, and the collected material can be output upon command. The **m4** macro processor maintains nine possible temporary files, numbered 1 through 9. If you use the built-in **divert** macro.

Macro	Description
divert (*Number***)**	Changes output stream to the temporary file *Number*.

The **m4** macro processor writes all output from the program after the **divert** function at the end of temporary file, *Number*. To return the output to the display screen, use either the **divert** or **divert(0)** function, which resumes the normal output process.

The **m4** macro processor writes all redirected output to the temporary files in numerical order at the end of processing. The **m4** macro processor discards the output if you redirect the output to a temporary file other than 0 through 9.

To bring back the data from all temporary files in numerical order, use the built-in **undivert** macro.

Macro	Description
undivert (*Number1***,** *Number2*... **)**	Appends the contents of the indicated temporary files to the current temporary file.

To bring back selected temporary files in a specified order, use the built-in **undivert** macro with arguments. When using the **undivert** macro, the **m4** macro processor discards the temporary files that are recovered and does not search the recovered data for macros.

The value of the **undivert** macro is not the diverted text.

You can use the **divnum** macro to determine which temporary file is currently in use.

Macro	Description
divnum	Returns the value of the currently active temporary file.

If you do not change the output file with the **divert** macro, the **m4** macro processor puts all output in a temporary file named 0.

Using system programs in a program

You can run any program in the operating system from a program by using the built-in **syscmd** macro. For example, the following statement runs the **date** program:

```
syscmd(date)
```

Using unique file Nnames

Use the built-in **maketemp** macro to make a unique file name from a program.

Macro	Description
maketemp (*String...nnnnn...String***)**	Creates a unique file name by replacing the characters *nnnnn* in the argument string with the current process ID.

For example, for the statement:

```
maketemp(myfilennnnn)
```

the **m4** macro processor returns a string that is `myfile` concatenated with the process ID. Use this string to name a

temporary file.

Using conditional expressions

Conditional expression evaluation allows process time determination of macro expressions.

Expression	Description
ifelse (*String1***, *String2***, *Argument1***, *Argument2***)**	If *String1* matches *String2*, returns the value of *Argument1*. Otherwise it returns *Argument2*.

The built-in **ifelse** macro performs conditional testing. In the simplest form:

```
ifelse(a, b, c, d)
```

compares the two strings a and b.

If a and b are identical, the built-in **ifelse** macro returns the string c. If they are not identical, it returns string d. For example, you can define a macro called `compare` to compare two strings and return `yes` if they are the same, or `no` if they are different, as follows:

```
define(compare, `ifelse($1, $2, yes, no)')
```

The quote characters prevent the evaluation of the **ifelse** macro from occurring too early. If the fourth argument is missing, it is treated as empty.

The **ifelse** macro can have any number of arguments, and therefore, provides a limited form of multiple-path decision capability. For example:

```
ifelse(a, b, c, d, e, f, g)
```

This statement is logically the same as the following fragment:

```
if(a == b) x = c;
else if(d == e) x = f;
else x = g;
return(x);
```

If the final argument is omitted, the result is null, so:

```
ifelse(a, b, c)
```

is c if a matches b, and null otherwise.

Manipulating strings

The macros in this section allow you to convert input strings into output strings.

Macro	Description
len	Returns the byte length of the string that makes up its argument

Thus:

```
len(abcdef)
```

is 6, and:

```
len((a,b))
```

is 5.

Macro	Description
dlen	Returns the length of the displayable characters in a string

Characters made up from 2-byte codes are displayed as one character. Thus, if the string contains any 2-byte, international character-support characters, the results of **dlen** will differ from the results of **len**.

Macro	Description
substr (*String***,** *Position***,** *Length***)**	Returns a substring of *String* that begins at character number *Position* and is *Length* characters long.

Using input, **substr (***s***,** *i***,** *n***)** returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. For example, the function:

```
substr(`now is the time',1)
```

returns the following string:

```
ow is the time
```

Macro	Description
index (*String1***,** *String2***)**	Returns the character position in *String1* where *String2* starts **(**starting with character number 0), or -1 if *String1* does not contain *String2*.

As with the built-in **substr** macro, the origin for strings is 0.

Macro	Description
translit (*String***,** *Set1***,** *Set2***)**	Searches *String* for characters that are in *Set1*. If it finds any, changes (transliterates) those characters to corresponding characters in *Set2*.

It has the general form:

```
translit(s, f, t)
```

which modifies `s` by replacing any character found in `f` by the corresponding character of `t`. For example, the function:

```
translit(`little', aeiou, 12345)
```

replaces the vowels by the corresponding digits and returns the following:

```
l3ttl2
```

If `t` is shorter than `f`, characters that do not have an entry in `t` are deleted. If `t` is not present at all, characters from `f` are deleted from `s`. So:

```
translit(`little', aeiou)
```

deletes vowels from string `little` and returns the following:

```
lttl
```

Macro	Description
dnl	Deletes all characters that follow it, up to and including the new-line character.

Use this macro to get rid of empty lines. For example, the function:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new-line at the end of each line that is not part of the definition. These new-line characters are passed to the output. To get rid of the new lines, add the built-in **dnl** macro to each of the lines.

```
define(N, 100) dnl
define(M, 200) dnl
define(L, 300) dnl
```

Debugging M4 macros

The macros in this section allow you to report errors and processing information.

Macro	Description
errprint (*String***)**	Writes its argument (*String*) to the standard error file

For example:

```
errprint (`error')
```

Macro	Description
dumpdef (`*MacroName*'... **)**	Dumps the current names and definitions of items named as arguments (`*MacroName*'...)

If you do not supply arguments, the **dumpdef** macro prints all current names and definitions. Remember to quote the names.

Additional m4 macros

A list of additional **m4** macros, with a brief explanation of each, follows:

Macro	Description
changecom (*l***,** *r* **)**	Changes the left and right comment characters to the characters represented by the *l* and *r* variables.
defn (*MacroName***)**	Returns the quoted definition of *MacroName*
en (*String***)**	Returns the number of characters in *String*.
m4exit (*Code***)**	Exits **m4** macro processor with a return code of *Code*.
m4wrap (*MacroName***)**	Runs macro *MacroName* at the end of **m4** macro processor .
popdef (*MacroName***)**	Replaces the current definition of *MacroName* with the previous definition saved with the **pushdef** macro.
pushdef (*MacroName***,** *Replacement***)**	Saves the current definition of *MacroName* and then defines *MacroName* to be *Replacement*.
sysval	Gets the return code from the last use of the **syscmd** macro.
traceoff (*MacroList***)**	Turns off trace for any macro in *MacroList*. If *MacroList* is null, turns off all tracing.

Macro	Description
traceon (*MacroName***)**	Turns on trace for macro *MacroName*. If *MacroName* is null, turns trace on for all macros.

Parent topic: General programming concepts

[Feedback]