m4 Macro Processor User's Manual

Table of Contents

Introduction

The COHERENT™ operating system macro processor **m4** is a powerful and flexible text processing tool. It allows the user to specify search strings (macro names) and strings to replace them (definitions) with a great degree of generality. Macros may take arguments, written in a natural functional notation.

m4 is useful as a front end for the COHERENT assembler **as**, which contains no built-in macro facility. It is also useful for higher-level languages like C, as well as for other applications requiring text replacement. **m4** includes powerful file manipulation, conditional decision making, substring selection and arithmetic capabilities, so it is useful for form processing.

The command line

m4 [*file* ... **]**

invokes **m4**. **m4** processes each specified input *file* in the order given on the command line and writes its output on the standard output. If no *file* is given, **m4** reads from the standard input. A *file* specified as '−' also indicates the standard input, allowing interactive input during file processing. **m4** reports any *file* which it cannot open and eliminates it from the input stream. As with other COHERENT commands, the optional output redirection specification '>*outfile*' on the command line redirects the output of **m4** to any desired file.

Definitions and Syntax

m4 copies text from its input stream to its output stream. When it finds a macro call, **m4** removes it from the input stream and evaluates it. The *replacement text* which results (called the *value* of the macro call) replaces the macro call on the input stream. **m4** processes the replacement text and then continues processing the remainder of its input stream.

'Text contained within quote characters' is *quoted* and all other text is *unquoted*. **m4** searches all unquoted text for *defined macro names*. A legal macro name is a string of alphanumeric characters (letters, digits, underscore '_'), the first of which is not a digit; **m4** recognizes the macro name only if it is surrounded by nonalphanumerics on both sides.

A *macro call* can be either a macro name or a macro name immediately followed by an argument set:

```
        macroname(arg1, ..., argn)
```

A set of arguments must start with a left parenthesis immediately following the macro name. The entire argument set must contain balanced unquoted parentheses; parentheses appearing in the argument text, if not quoted, must always be in pairs. A single open or close parenthesis may be passed by quoting it, e.g. \`(' or \`)'.

A comma which is not inside quotes or inside an inner set of unquoted parentheses separates *arguments* in the argument set. **m4** strips each argument of leading unquoted spaces, tabs, and newlines. It processes the text of each argument in the same manner as ordinary text, so it removes, evaluates, and replaces any recognized macro calls *before* it stores the argument text for possible use within the replacement text; to actually pass a macro name or an entire macro call as an argument, it must be quoted. **m4** stores the values of the first nine arguments for possible use in the replacement text. It processes arguments after the ninth but throws away the results.

m4 does not search quoted text for macro calls. Instead, it removes the quote marks and passes the contained text unchanged. Quotes are nestable; that is, quoted text may contain other sets of quotes. **m4** removes only the outer level of quotes each time it reads a piece of text. This facilitates delaying macro expansion in text until the second (or later) time **m4** reads the text.

Predefined **m4** macros perform various functions. The remainder of this document describes the predefined macros in detail. The final section is a summary, containing an alphabetized list and brief description of each predefined macro.

Defining Macros

The macro call

```
        define(`name', `definition')
```

defines a macro *name* with the replacement text *definition*. **m4** replaces every subsequent unquoted occurrence of *name* with the *definition*, as described above. For example, the **m4** input

```
        define(`her', `COHERENT')
        To know, know, know her
        Is to love, love, love her...
```

produces the output

```
        To know, know, know COHERENT
        Is to love, love, love COHERENT...
```

The *name* argument to **define** should usually be quoted. If it is not quoted and it is being redefined, **m4** sees its old *definition* as the first argument to **define**, which will not have the intended effect. Similarly, the *definition* should be quoted if macro names which occur in it should not be replaced.

Any legal macro name may be the first argument of a **define**. If a predefined macro is redefined, its original function is lost and cannot be recovered.

As noted above, **m4** recognizes a macro name only if it is surrounded by nonalphanumerics. For example,

```
        define(`her', `COHERENT')
        Coherent software is reliable software.
```

produces the output

```
        Coherent software is reliable software.
```

m4 does not recognize the characters **her** in the word **Coherent** as a macro name.

The value of the **define** macro is the null or empty string (the string which contains no characters). In other words, **m4** puts nothing (the null string) back on its input stream when it processes a **define** call.

Like predefined macros, user-defined macros may take arguments. **m4** replaces the string *$n* in the macro definition with the value of the *n*th argument, where *n* is a digit (1 to 9). It replaces $0 with the macro name. If the argument set contains fewer than *n* arguments, **m4** replaces *$n* with the null string. **m4** uses functional notation to specify argument sets. Unlike a normal function, however, an **m4** macro does not require a fixed number of arguments. The same macro may be called with or without an argument set, or with argument sets containing different numbers of arguments.

The following macro concatenates its arguments:

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

Then

```
      cat(one, `two', ``three'', `four, four ',
         five(also,),,seven)
```

becomes

```
        onetwo`three'four, four five(also,)seven
```

A more complex definition is:

```
        define(`comma', ``$0 (which looks like `,')'')
```

This turns each subsequent unquoted occurrence of

```
        comma
```

into

```
        comma (which looks like `,')
```

Two sets of quotes around the replacement text are necessary. When **m4** reads this call to macro **define**, the resultant argument text is:

```
        comma
```

for the *name* and

```
        `$0 (which looks like `,')'
```

for the *definition*. When **m4** sees the text

```
        comma that is not quoted
```

it evaluates and replaces the now-defined macro name **comma** to produce the text

```
        `comma (which looks like `,')' that is not quoted
```

on the *input* stream. Since **comma** appears inside a set of quotation marks, **m4** will not treat it as a macro name. For the same reason, the string ',' also passes through unmodified. The final output is:

```
        comma (which looks like `,') that is not quoted
```

The predefined macro **dumpdef** without arguments returns the names and definitions of all defined macros. For each macro, it returns its quoted name, followed by a tab character, followed by its quoted definition; no definition is given for a predefined macro. With arguments,

```
        dumpdef(names)
```

returns the quoted definition of each macro name specified as an argument.

The predefined macro

```
        undefine(`name')
```

removes a macro definition. As noted for **define** above, the argument must be quoted to have the desired effect. **undefine** ignores arguments which are not defined macro names. The value of the **undefine** call is the null string. If a predefined macro is undefined, its original function cannot be recovered.

Input Control

The predefined macro **changequote** changes the quote characters, which are initially ` and '. For example:

```
        changequote( {, })
```

makes the quote characters the left and right braces, removing the effect of the previously defined quote characters. Missing arguments default to ` for open quote and ' for close quote; thus, **changequote** without arguments restores the original quote characters ` and '. If the arguments are identical, the nesting ability of quotes is temporarily lost. Instead, the first instance of the new quote character turns on quoting and the next instance turns off quoting. The value of the **changequote** call is the null string.

The predefined macro **dnl** (delete to newline) "eats" all characters from the input stream up to and including the next newline and returns the null string. It is particularly useful in a string of **define** macro calls. Although **m4** replaces each **define** by the null string, newlines often separate macro definitions, and **m4** copies the newlines to the output stream unchanged. Two ways of using **dnl** are:

```
define(this, that)dnl
define(something, else)dnl

dnl(define(this, that), define(something, else))
```

The first examples use **dnl** without arguments. The final example uses **dnl** with an argument set, which **m4** processes (performing each **define**) and subsequently ignores. The following section describes an alternative (and generally preferable) method of eliminating extraneous newlines in a sequence of **define** calls.

m4 includes two decision-making macros. The predefined macro **ifdef** returns one of two values each time it is called:

```
ifdef(`name', defvalue, undefvalue)
```

If the first argument is a legal macro name and is defined, the replacement text is *defvalue*; otherwise, the replacement text is *undefvalue*.

ifelse is a more general decision-making macro. The basic call takes four arguments:

```
ifelse(arg1, arg2, arg3, arg4)
```

ifelse compares *arg1* and *arg2*. If they are equal, **ifelse** returns *arg3*. If not, it returns *arg4*. More than four arguments cause another comparison:

```
ifelse(arg1, arg2, arg3, arg4, arg5, arg6, arg7)
```

As with the form above, this call of **ifdef** compares *arg1* and *arg2* and returns *arg3* if they are equal. Otherwise, it compares *arg4* and *arg5*. It returns *arg6* if they are equal, *arg7* otherwise. If more than seven arguments are present and *arg4* and *arg5* are not equal, **ifelse** compares *arg7* and *arg8*. It returns *arg9* if they are equal and the null string otherwise.

In addition to each *file* specified in the command line, any other accessible file may be included in the input stream with the predefined macro

```
include(file)
```

m4 replaces this macro call on the input stream with the entire contents of the specified *file*. If *file* cannot be accessed, **include** causes a fatal error; **m4** prints an error message and exits. The alternative predefined macro

```
sinclude(file)
```

functions exactly like **include**, except that it does not print an error message and stop processing if *file* is inaccessible.

Output Control

m4 maintains ten output streams, numbered zero through nine. Stream 0 is the standard output, where **m4** normally directs its output. Streams 1 through 9 are temporary files. The predefined macro

```
divert(n)
```

diverts output away from stream 0, appending it instead to stream *n*. Any *n* outside the range 0–9 causes output to be thrown away until the next **divert** call. **divert** without any arguments or with a nonnumeric argument is equivalent to **divert(0)**. The value of a **divert** call is the null string.

The preceding section described the use of **dnl** to eliminate extraneous newlines on the output stream when processing a sequence of **define** calls. A more readable method of eliminating the newlines is to precede the definitions with **divert(−1)** and follow them with **divert**. **m4** then diverts the extraneous newlines to the nonexistent stream −1.

The predefined macro

```
undivert(streams)
```

fetches text diverted to one or more temporary streams. It appends the text from the specified *streams* in the given order to the *current* output stream. **m4** will not allow diverted text to be undiverted back to the same stream. **undivert** with no arguments undiverts all diversions in numerical order. The value of **undivert** is the null string; undiverted text is *not* scanned for macro calls, but is simply moved from one place to another. **m4** automatically undiverts all diversions in numerical order to the standard output (stream 0) at the end of processing.

The predefined macro **divnum** returns the current diversion number.

The predefined macro

```
        errprint(message)
```

sends the given *message* to the standard error stream. The value of **errprint** is the null string.

String Manipulation

The predefined macro

```
        substr(string, start, count)
```

returns a substring of a string of characters. The first argument *string* can be anything. The second argument *start* is a number giving the starting position of the desired substring in *string*. Position 0 is the leftmost character of *string*, position 1 is the next character to the right, and so on. If *start* is negative, the orientation switches to the right; position −1 is the rightmost character of *string*, position −2 is the character to its left, and so on. The third argument *count* specifies the length and direction of the substring. Zero returns the null string. A positive *count* returns a substring consisting of the character addressed by *start* and *count*−1 characters to the right of it. A negative number does the same thing, but to the left. If *count* is omitted, it is assumed to be of the same sign as *start* and large enough to extend to the end of *string* in that direction. If *start* is omitted, it is assumed to be 0 if *count* is positive or omitted, or −1 if *count* is negative. For example:

```
        define(`alpha', `abcdefghijklmnopqrstuvwxyz')
        substr(alpha, , )
```

returns

```
        abcdefghijklmnopqrstuvwxyz
```

Here both *start* and *count* are omitted and are therefore assumed to be 0 and 26, respectively.

```
        substr(alpha, 0, 6)
        substr(alpha, , 6)
```

both return

```
        abcdef
```

Similarly,

```
        substr(alpha, , -6)
        substr(alpha, 21, )
```

both return

```
        uvwxyz
```

Finally,

```
        substr(alpha, -6, )
        substr(alpha, 0, 21)
```

both return

```
        abcdefghijklmnopqrstu
```

The predefined macro

```
        translit(string, characters, replacements)
```

transliterates single characters within a string. It returns *string* with every occurrence of a character specified in *characters* replaced with the corresponding character from *replacements*. If there is no corresponding character, **translit** simply deletes the character. For example:

```
        translit(alpha, aeiouy, *+-=/)
```

returns

```
        *bcd+fgh-jklmn=pqrst/vwxz
```

Numeric Manipulation

m4 can simulate variables typical of most programming languages by using **define** as the assignment operator. Whenever the defined macro name appears unquoted, **m4** immediately replaces it by its numeric value.

The predefined macros **incr** and **decr** return their argument incremented or decremented by 1. Thus,

```
define(`x', 1234)
incr(x)
```

returns

```
1235
```

incr and **decr** assume an argument which is omitted or not a valid number to be 0.

More generally, the predefined macro

eval(*expression*)

evaluates an integer-value arithmetic *expression* and returns the resulting value. The operators available, in order of decreasing precedence, are:

```
( )                   (parentheses for grouping)
+ -                   (unary plus, negation)
^ **                  (exponentiation)
* / %                 (multiplication, division, modulus)
+ -                   (addition, subtraction)
> < >= <= == !=   (comparisons)
!                     (logical negation)
&& &                  (logical and)
|| |                  (logical or)
```

The comparisons and logical operators return either 0 (false) or 1 (true). **eval** performs all arithmetic in long integers. **eval** reports an error if its argument is not a well-formed expression.

The predefined macro

len(*string*)

returns a numeric value corresponding to the length of *string*.

The predefined macro

index(*string*, *pattern*)

returns a numeric value corresponding to the first position where *pattern* appears in *string*. If it does not appear, **index** returns −1. Both *pattern* and *string* may be arbitrary strings of any length.

The following example defines a macro **repeat** which repeats its first argument the number of times specified by its second argument.

```
define(`repeat',
        `ifelse(eval($2<=0),1,,`repeat($1,decr($2) )'$1)')
```

The definition is recursive; that is, **repeat** calls itself within its own definition. The entire definition is quoted to defer the evaluation of **ifelse** from when **m4** encounters the definition to when it encounters a **repeat** macro call. Similarly, the recursive **repeat** call is quoted to defer its evaluation within the **ifelse**. **eval** checks if the first argument is less than or equal to 0; if so, it returns 1 (true) and **ifelse** returns the null string. Otherwise, **decr** decrements the count so each successive recursive call has a smaller second argument, and each call appends a copy of the first argument to the previous result. For example:

```
repeat(`Ho! ',3)
```

produces

```
Ho! Ho! Ho!
```

COHERENT System Interface

The predefined macro

```
maketemp(string)
```

returns *string* with its last six characters (normally **XXXXXX**) replaced by a five digit number corresponding to the current process id and a unique single letter. It is the same as the C library routine **mktemp**. It returns the null string if its argument is less than six characters long.

The predefined macro

```
syscmd(command)
```

performs the given COHERENT *command* and returns the null string. It is the same as the C library routine **system**.

A common use of **syscmd** is to create a file which **m4** subsequently reads with an **include**. For example, to get the output from the COHERENT **date** command:

```
define(`tempfile', maketemp(/tmp/m4XXXXXX))
define(`get_date',
        `syscmd(date >tempfile)'`include(tempfile)')
```

In subsequent input, **m4** replaces each occurrence of **get_date** with the system date information. The definition of **tempfile** is unquoted, so **m4** executes the **maketemp** call only once (when it processes the define), and it creates only one temporary file. On the other hand, the definition of **getdate** is quoted, so **m4** executes **syscmd** and **include** to get the current time and date each time it processes a **getdate** call. The temporary file should be removed with

```
syscmd(rm tempfile)
```

at the end of the **m4** program.

The following example is more complex. It defines a macro **save** which appends a macro definition to a file.

```
define(`save',`syscmd(`cat >>$2 <<\#
define(`$1',`dumpdef(`$1')')
#
')')
```

The arguments to **define** are the *name*

```
save
```

and the *definition*

```
syscmd(`cat >>$2 <<\#
define(`$1',`dumpdef(`$1')')
#
')
```

A typical call of this macro is:

```
save(`sample',`defs.m4')
```

which saves the macro definition of **sample** in a COHERENT file **defs.m4** containing macro definitions. When **m4** processes this call, the argument of **syscmd** becomes

```
cat >>efs.m4 <<\#
define(`sample',
```

followed by the definition of **sample** returned by **dumpdef**, followed by

```
)
#
```

Then **syscmd** executes the COHERENT **cat** command to append the here document delimited by **#** to the macro definition file **defs.m4**. The leading **#** delimiter of the here document is quoted with **** to prevent interpretation by the COHERENT shell. Because the **save** macro uses the character **#** to delimit the here document, it does not work correctly for macro definitions containing **#**. For example,

```
save(`save',`defs.m4')
```

does not work as expected.

Errors

m4 reports all errors to the standard error stream. An error produces a line of the form

```
m4: line: message
```

where *line* is a decimal line number and *message* describes the error. For example, the error message

```
m4:  7:  illegal macro name: ab*c
```

indicates an attempt to **define** a macro with the illegal macro name **ab*c** in line 7 of the input stream.

The following error messages may occur:

```
cannot open file
eval: invalid expression
eval: missing or unknown operator
eval: missing value
illegal macro name: name
out of space
/tmp open error
unexpected EOF
```

The *file* or *name* will be the file name or macro name which caused the error, or {NULL} if the required argument is omitted.

m4 does not recognize (and therefore does not report) the most common of m4 errors, namely invoking recursive macro definitions which never terminate. A simple example is the definition

```
define(`recursive', `recursive')
```

When **m4** subsequently encounters a call of **recursive** in its input stream, it replaces it on the input stream with its definition. Since the definition is another **recursive** call, **m4** replaces it in turn with its definition; the process never terminates. More complicated examples may involve many macro definitions and may be difficult to discover. If **m4** enters an endless loop, the user can terminate it from the keyboard by typing the interrupt character (normally DELETE) or the kill character (normally <ctrl-\>). If **m4** enters an endless loop while being run in the background, the user can terminate it with the **kill** command.

Summary

The following alphabetized list gives the name and function of each **m4** predefined macro. Square brackets '[ ]' indicate that the enclosed item is optional. Ellipses '...' indicate optional additional occurrences of the preceding item.

changequote[([*openquote*],[*closequote*])]
: Changes the quote characters. Missing arguments default to ` for open or ' for close. Quotes will not nest if they are defined to be the same character. Value is null.

decr[(*number*)]
: Decrements given *number* (default: 0) by one and returns resulting value.

define(*macroname*,*definition*)
: Defines or redefines any macro. If a predefined macro is redefined, its original definition is irrecoverably lost. Value is null.

divert[(*n*)]
: Redirects output to output stream *n* (default: 0, the standard output). The standard output is 0, and 1 – 9 are maintained as temporary files. Any other *n* results in output being thrown away until the next **divert** macro. Value is null.

divnum
: Value is current output stream number.

dnl
: Delete to newline: removes all characters from the input stream up to and including the next newline. Value is null.

dumpdef[(*macronames*)]
: Value is quoted definitions of all *macronames* specified, or names and definitions of all defined macros if no arguments.

errprint(*text*)
: Prints *text* on standard error file. Value is null.

eval(*expression*)
: Value is a number which is the value of evaluated *expression*. Recognizes, in order of decreasing precedence: parentheses, ^ **, unary + –, * / %, binary + –, relations, and logicals. Arithmetic is performed in longs.

ifdef(*macroname*,*defvalue*,*undefvalue*)
: Returns *defvalue* if *macroname* is defined and *undefvalue* if not.

ifelse(*arg1*,*arg2*,*arg3*...)
: Compares *arg1* and *arg2*. If they are the same, returns *arg3*. If not, and *arg4* is the last argument, returns *arg4*. Otherwise, the process repeats, comparing *arg4* and *arg5*,

and so on. Like other **m4** macros, takes a maximum of nine arguments.

include(*file*)
Value is the entire contents of the *file* argument. If *file* is not accessible, a fatal error results.

incr[(*number*)]
Increments given *number* (default: 0) by one and returns resulting value.

index(*text*,*pattern*)
Value is a number corresponding to position of *pattern* in *text*. If *pattern* does not occur in *text*, value is −1.

len(*text*)
Value is a number corresponding to length of *text*.

maketemp(*text*)
Value is *text* with last six characters, usually **XXXXXX**, replaced with current process id and a single letter. Same as system call **mktemp**.

sinclude(*file*)
Value is the entire contents of the *file* argument. If the *file* is not accessible, returns null and processing continues.

substr(*text*[,*start*[,*count*]])
Value is a substring of *text*. The *start* position may be left-oriented (nonnegative) or right-oriented (negative). The *count* specifies how many characters to the right (if positive) or to the left (if negative) to return. If absent, it is assumed to be large and of the same sign as *start*. If *start* is omitted, it is assumed to be 0 if *count* is positive or omitted, or −1 if *count* is negative.

syscmd(*command*)
Passes the given *command* to the shell **sh** for execution. Value is null. Same as system call **system**.

translit(*text*,*characters*[,*replacements*])
Replaces each occurrence in *text* of given *characters* with the corresponding characters from *replacements*. If the *replacements* argument is absent or too short, null replaces selected *characters*. Value is *text* with specified replacements.

undefine(*macroname*)
Removes macro definition. If a predefined macro is redefined, its original definition is irrecoverably lost. Value is null.

undivert[(*stream*[,...])]
Dumps each specified *stream* onto the current output stream. With no arguments, **undivert** dumps all output streams in numeric order. **m4** will not dump any output stream onto itself. At the end of processing, **m4** automatically dumps all diverted text to standard output in numeric order. Value is null.

Index