Introduction to the m4 Macro Processor

m4 is a macro processor for the COHERENT system. It is a powerful and flexible text processing tool. You can tell it, with a great degree of generality, to search for macro names and replace them with other strings. Macros can also take arguments.

m4 provides a useful front end for programming languages such as fourth-generation lanaguages (4GLs) which commonly have no built-in macro facility. **m4** also has powerful facilities for manipulating files, making decisions conditionally, selecting substrings, and performing arithmetic, so it is useful for processing forms.

The command

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

invokes **m4**. **m4** reads each *file* in the order given on the command line; if no *file* is given, **m4** reads from the standard input. The *file* '-' also indicates the standard input; this allows you to perform interactive input while **m4** is processing files. **m4** reports any *file* that it cannot open, and eliminates it from the input stream.

m4 writes its output to the standard output stream. As with other COHERENT commands, the optional output redirection specification **>***outfile* on the command line redirects the output into *outfile.* To leave **m4**, type **<ctrl-D>**.

Definitions and Syntax

m4 reads text one line at a time from its input stream. When it reads a line of text, it scans the line for a macro that you have defined. 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 nonalphanumeric characters (i.e., spaces or newline characters) on both sides.

When **m4** finds a macro, it removes it from the input stream and replaces it with its definition. It then writes the resulting modified text (called *replacement text*), onto the input stream. **m4** then reads another line from the input stream, and continues processing.

Text that is contained within single quotation marks is quoted (i.e., is contained between a grave mark ` on the left and an apostrophe ' on the right). All other text is *unquoted.* **m4** searches only unquoted text for macros.

A *macro call* can be either a macro or a macro immediately followed by a set of arguments:

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

A set of arguments must start with a left parenthesis that follows the macro immediately (i.e., no space can come between the macro and the left parenthesis). The entire argument set must be enclosed by balanced, unquoted parentheses: parentheses may appear within the text of an argument, but they must always come in balanced pairs. A single left or right parenthesis may be passed by quoting it, e.g. `(' or `)'.

Arguments are separated by commas that are neither within apostrophes nor within an inner set of unquoted parentheses. **m4** strips from each argument all leading unquoted spaces, tabs, and newlines. It processes the text of each argument in the same manner that it processes ordinary text; that is, it removes, evaluates, and replaces any recognized macro calls *before* it stores the argument text for possible use within the replacement text. If you wish to 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 macros. Instead, it removes the quotation marks and copies the text to the standard output unchanged. Quotes can be nested; that is, quoted text can contain other blocks of quoted text. **m4** removes only the outermost level of quotation marks each time it reads a piece of quoted text. This aids in delaying macro expansion in text until the second (or later) time the text is read by **m4**.

m4 includes numerous predefined macros, which perform various functions. The remainder of this document describes the predefined macros in detail. The Lexicon entry for **m4** summarizes each predefined macro.

Defining Macros

The macro

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

defines a macro *name* and its replacement text *definition*. **m4** replaces every subsequent unquoted occurrence of *name* with *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 ...
```

name 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, *definition* should be quoted if the macro names that occur in it should not be replaced.

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

As noted above, **m4** recognizes a macro name only if it is surrounded by non-alphanumeric characters. 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

```
onetwothreefour, 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 quotation marks 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. Because **comma** appears inside a set of quotation marks, **m4** does 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
```

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

```
dumpdef(name)
```

returns the quoted definition of each macro name that appears 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. For example:

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

makes the quote characters the left and right braces. It also removes the effect of the previously defined quotation characters. Missing arguments default to ` for open quotation and ' for close quotation. Thus, **changequote** without arguments restores the original quote characters ` and '. If the arguments are identical, the nesting ability of quotation marks 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: **ifdef** and **ifelse**.

ifdef checks whether a macro is defined. It has the following form:

```
ifdef(macro,defvalue,undefvalue)
```

If *macro* is defined, **ifdef** returns *defvalue*; otherwise, it returns *undefvalue*.

ifelse compares pairs of arguments. It has the following form:

```
ifelse(arg1,arg2,arg3, ... , arg9)
```

ifelse compares *arg1* with *arg2*. If they are the same, it returns *arg3*. If not, and if *arg4* is the last argument, it returns *arg4*. Otherwise, it repeats the process, comparing *arg4* with *arg5*, and so on. Like other **m4** macros, this takes a maximum of nine arguments.

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 to 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** does 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.

To illustrate the use of **divert** and **undivert**, invoke **m4** and type:

```
define(`count', $1$2)
```

And to see what macro **count** does, type:

```
count(one, three)
```

The output on the screen reads:

```
onetwo
```

Now type:

```
divert(1)
```

This diverts device 1 (the standard output) into a temporary file. Now type:

```
count(one, three)
```

Nothing appears on the screen. **divert** sent the output of the macro **count(one, three)** into a temporary file. Thus, the output is not lost, as you might have thought. To demonstrate the existence of that output, type:

```
divert
```

to reset the standard output to be the screen. See for yourself. Now, when you type

```
count(one, four)
```

m4 replies on the screen:

```
onefour
```

As you can see, the standard output is again directed to the screen. To retrieve the diverted output of **count(one, three)**, and send it to the screen, type:

```
undivert(1)
```

which produces:

```
onethree
```

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:

```
define(liquorjugs, `pack my box with five dozen liquor jugs')
translit(liquorjugs, aeiou, 1234)
```

returns:

```
p1ck my b4x w3th f3v2 d4z2n l3q4r jgs
```

Numeric Manipulation

m4 can simulate the long integer 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
```

Note that **incr** and **decr** do not change the value of the simulated variable **x**, or of any other variable. They return only that value plus or minus 1; **x** itself retains its value of **1234**.

incr and **decr** initialize to zero all arguments that are omitted or not a valid number. Thus, the example

```
incr(a34/87)
```

returns **1**; but

```
incr(123.67)
```

returns **124**. As you can see, **incr** truncates floating-point numbers. The same applies to a variable that you have **define**d to have a floating-point value.

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** that 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!
```

From this example, you can see that the lowered value of the second argument — generated by the macro **decr**— is "kept in mind" successively. Nevertheless, **decr** and **incr** never change the value of a variable. For example, consider:

```
define(`turns', 10)
```

We now have a variable called **turns** whose value is ten. Typing

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

produces:

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

Within **repeat**, **decr** lowered the current value of the second argument (i.e., **turns**), until it becomes zero. But when we type

```
turns
```

we see:

```
10
```

As you can see, the value of **turns** remained ten, despite that variable's having been used in a **decr** statement.

COHERENT System Interface

The predefined macro

```
maketemp(string)
```

creates a unique file name for a temporary file. *string* is a six-character string that is normally initialized to **XXXXXX**; **maketemp** replaces all of the **X**s with a pattern of six numerals that form a unique file name in the directory where temporary files are being written. 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 call to **get_date**. 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')`)
#
')
```

(Note that the body of macro **syscmd** uses the shell operator **<<** to create a “here document”. For more information on here documents, see the tutorial *Intorducing sh, the Bourne Shell.*) 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 >>defs.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 **save** 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.

Note that you can only use **save** when you run **m4** interactively — you cannot use it in a script. Furthermore, **save** does not always save a definition literally. For example:

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

saves the **tempfile** definition in **defs.m4** as:

```
define(`tempfile', `/tmp/m400074a') #
```

where, as you can see, the **XXXXXX** has been replaced with a hexadecimal number (which may differ from the one you). Likewise, the definition of **get_date** will look like this:

```
define(`get_date', `syscmd(date >tempfile)include(tempfile)') #
```

To load a saved definition into **m4**, simply type **m4** at the shell's command-line prompt to invoke it interactively; and then type:

```
sinclude(defs.m4)
```

From now on, you can use any definition that you had saved into file **defs.m4**.

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 that 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. Because the definition is another call to **recursive**, **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, you can terminate it from the keyboard by typing the interrupt character (normally **<ctrl-C>**) or the kill character (normally **<ctrl-\>**). If **m4** enters an endless loop while being run in the background, you can terminate it with the **kill** command.

For More Information

The Lexicon entry for **m4** gives a summary of its functions and options.