m4

COLLABORATORS

	TITLE : m4	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 15, 2023	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

m4

1.1 m4.guide

```
                GNU m4
******
```

GNU m4 is an implementation of the traditional UNIX macro processor. It is mostly SVR4 compatible, although it has some extensions (for example, handling more than 9 positional parameters to macros). m4 also has builtin functions for including files, running shell commands, doing arithmetic, etc. Autoconf needs GNU m4 for generating configure scripts, but not for running them.

GNU m4 was originally written by Ren'e Seindal, with subsequent changes by Franc,ois Pinard and other volunteers on the Internet. All names and email addresses can be found in the file THANKS from the GNU m4 distribution.

This is release 1.4. It is now to be considered stable, future releases are only meant to fix bugs, increase speed, or improve documentation. However...

An experimental feature, which would improve m4 usefulness, allows for changing the syntax for what is a word in m4. You should use:

```
./configure --enable-changeword
```

if you want this feature compiled in. The current implementation slows down m4 considerably and is hardly acceptable. So, it might go away, do not count on it yet.

1.2 m4.guide/Preliminaries

```
                Introduction and preliminaries
******************************
```

This first chapter explains what is GNU m4, where m4 comes from, how to read and use this documentation, how to call the m4 program and how to report bugs about it. It concludes by giving tips for reading the remainder of the manual.

The following chapters then detail all the features of the m4 language.

1.3 m4.guide/Intro

Introduction to m4
==========

m4 is a macro processor, in the sense that it copies its input to the output, expanding macros as it goes. Macros are either builtin or user-defined, and can take any number of arguments. Besides just doing macro expansion, m4 has builtin functions for including named files, running UNIX commands, doing integer arithmetic, manipulating text in various ways, recursion, etc... m4 can be used either as a front-end to a compiler, or as a macro processor in its own right.

The m4 macro processor is widely available on all UNIXes. Usually, only a small percentage of users are aware of its existence. However, those who do often become commited users. The growing popularity of GNU Autoconf, which prerequires GNU m4 for generating the configure scripts, is an incentive for many to install it, while these people will not themselves program in m4. GNU m4 is mostly compatible with the System V, Release 3 version, except for some minor differences. See

Compatibility
 for more details.

Some people found m4 to be fairly addictive. They first use m4 for simple problems, then take bigger and bigger challenges, learning how to write complex m4 sets of macros along the way. Once really addicted, users pursue writing of sophisticated m4 applications even to solve simple problems, devoting more time debugging their m4 scripts than doing real work. Beware that m4 may be dangerous for the health of compulsive programmers.

1.4 m4.guide/History

```
Historical references
=====================

   The historical notes included here are fairly incomplete, and not
authoritative at all.  Please knowledgeable users help us to more
properly write this section.

   GPM has been an important ancestor of m4.  See C. Stratchey: "A
General Purpose Macro generator", Computer Journal 8,3 (1965), pp. 225
ff.  GPM is also succintly described into David Gries classic "Compiler
Construction for Digital Computers".

   While GPM was pure, m4 was meant to deal more with the true
intricacies of real life: macros could be recognized with being
pre-announced, skipping whitespace or end-of-lines was made easier,
more constructs were builtin instead of derived, etc.

   Originally, m4 was the engine for Rational FORTRAN preprocessor,
that is, the ratfor equivalent of cpp.
```

1.5 m4.guide/Invoking m4

```
                Invoking m4
===========

   The format of the m4 command is:

     m4 [OPTION...] [MACRO-DEFINITIONS...] [INPUT-FILE...]

   All options begin with -, or if long option names are used, with a
--.  A long option name need not be written completely, and
unambigous prefix is sufficient.  m4 understands the following options:

--version
     Print the version number of the program on standard output, then
     immediately exit m4 without reading any INPUT-FILES.

--help
     Print an help summary on standard output, then immediately exit m4
     without reading any INPUT-FILES.

-G
--traditional
     Suppress all the extensions made in this implementation, compared
     to the System V version.  See
                Compatibility
                , for a list of these.

-E
--fatal-warnings
     Stop execution and exit m4 once the first warning has been issued,
```

considering all of them to be fatal.

-dFLAGS
--debug=FLAGS
Set the debug-level according to the flags FLAGS. The debug-level controls the format and amount of information presented by the debugging functions. See
Debug Levels
for more details on the
format and meaning of FLAGS.

-lNUM
--arglength=NUM
Restrict the size of the output generated by macro tracing. See

Debug Levels
for more details.

-oFILE
--error-output=FILE
Redirect debug and trace output to the named file. Error messages are still printed on the standard error output. See
Debug Output
for more details.

-IDIR
--include=DIR
Make m4 search DIR for included files that are not found in the current working directory. See
Search Path
for more details.

-e
--interactive
Makes this invocation of m4 interactive. This means that all output will be unbuffered, and interrupts will be ignored.

-s
--synclines
Generate synchronisation lines, for use by the C preprocessor or other similar tools. This is useful, for example, when m4 is used as a front end to a compiler. Source file name and line number information is conveyed by directives of the form #line LINENUM "FILENAME", which are inserted as needed into the middle of the input. Such directives mean that the following line originated or was expanded from the contents of input file FILENAME at line LINENUM. The "FILENAME" part is often omitted when the file name did not change from the previous directive.

Synchronisation directives are always given on complete lines per themselves. When a synchronisation discrepancy occurs in the middle of an output line, the associated synchronisation directive is delayed until the beginning of the next generated line.

-P
--prefix-builtins
Internally modify all builtin macro names so they all start with

the prefix m4_. For example, using this option, one should write m4_define instead of define, and m4___file__ instead of __file__.

-WREGEXP
--word-regexp=REGEXP
Use an alternative syntax for macro names. This experimental option might not be present on all GNU m4 implementations. (see

Changeword
).

-HN
--hashsize=N
Make the internal hash table for symbol lookup be N entries big. The number should be prime. The default is 509 entries. It should not be necessary to increase this value, unless you define an excessive number of macros.

-LN
--nesting-limit=N
Artificially limit the nesting of macro calls to N levels, stopping program execution if this limit is ever exceeded. When not specified, nesting is limited to 250 levels.

The precise effect of this option might be more correctly associated with textual nesting than dynamic recursion. It has been useful when some complex m4 input was generated by mechanical means. Most users would never need this option. If shown to be obtrusive, this option (which is still experimental) might well disappear.

This option does not have the ability to break endless rescanning loops, while these do not necessarily consume much memory or stack space. Through clever usage of rescanning loops, one can request complex, time-consuming computations to m4 with useful results. Putting limitations in this area would break m4 power. There are many pathological cases: define(‘a’, ‘a’)a is only the simplest example (but see
Compatibility
). Expecting GNU m4 to detect these would be a little like expecting a compiler system to detect and diagnose endless loops: it is a quite hard problem in general, if not undecidable!

-Q
--quiet
--silent
Suppress warnings about missing or superflous arguments in macro calls.

-B
-S
-T
These options are present for compatibility with System V m4, but do nothing in this implementation.

```
-NN
--diversions=N
     These options are present only for compatibility with previous
     versions of GNU m4, and were controlling the number of possible
     diversions which could be used at the same time.  They do nothing,
     because there is no fixed limit anymore.
```

Macro definitions and deletions can be made on the command line, by using the -D and -U options. They have the following format:

```
-DNAME
-DNAME=VALUE
--define=NAME
--define=NAME=VALUE
     This enters NAME into the symbol table, before any input files are
     read.  If  =VALUE is missing, the value is taken to be the empty
     string.  The VALUE can be any string, and the macro can be defined
     to take arguments, just as if it was defined from within the input.

-UNAME
--undefine=NAME
     This deletes any predefined meaning NAME might have.  Obviously,
     only predefined macros can be deleted in this way.

-tNAME
--trace=NAME
     This enters NAME into the symbol table, as undefined but traced.
     The macro will consequently be traced from the point it is defined.

-FFILE
--freeze-state FILE
     Once execution is finished, write out the frozen state on the
     specified FILE (see
                Frozen files
                ).

-RFILE
--reload-state FILE
     Before execution starts, recover the internal state from the
     specified frozen FILE (see
                Frozen files
                ).
```

The remaining arguments on the command line are taken to be input file names. If no names are present, the standard input is read. A file name of - is taken to mean the standard input.

The input files are read in the sequence given. The standard input can only be read once, so the filename - should only appear once on the command line.

1.6 m4.guide/Bugs

```
Problems and bugs
=================

   If you have problems with GNU m4 or think you've found a bug, please
report it.  Before reporting a bug, make sure you've actually found a
real bug.  Carefully reread the documentation and see if it really says
you can do what you're trying to do.  If it's not clear whether you
should be able to do something or not, report that too; it's a bug in
the documentation!

   Before reporting a bug or trying to fix it yourself, try to isolate
it to the smallest possible input file that reproduces the problem.
Then send us the input file and the exact results m4 gave you.  Also
say what you expected to occur; this will help us decide whether the
problem was really in the documentation.

   Once you've got a precise problem, send e-mail to (Internet)
bug-gnu-utils@prep.ai.mit.edu or (UUCP)
mit-eddie!prep.ai.mit.edu!bug-gnu-utils.  Please include the
version number of m4 you are using.  You can get this information with
the command m4 --version.

   Non-bug suggestions are always welcome as well.  If you have
questions about things that are unclear in the documentation or are
just obscure features, please report them too.
```

1.7 m4.guide/Manual

```
Using this manual
=================

   This manual contains a number of examples of m4 input and output,
and a simple notation is used to distinguish input, output and error
messages from m4.  Examples are set out from the normal text, and shown
in a fixed width font, like this

     This is an example of an example!

   To distinguish input from output, all output from m4 is prefixed by
the string =>, and all error messages by the string error-->.  Thus

     Example of input line
     =>Output line from m4
     error-->and an error message

   As each of the predefined macros in m4 is described, a prototype
call of the macro will be shown, giving descriptive names to the
arguments, e.g.,

     regexp(STRING, REGEXP, opt REPLACEMENT)

   All macro arguments in m4 are strings, but some are given special
interpretation, e.g., as numbers, filenames, regular expressions, etc.
```

The opt before the third argument shows that this argument is optional--if it is left out, it is taken to be the empty string. An ellipsis (...) last in the argument list indicates that any number of arguments may follow.

This document consistently writes and uses builtin, without an hyphen, as if it were an English word. This is how the builtin primitive is spelled within m4.

1.8 m4.guide/Syntax

```
                Lexical and syntactic conventions
**********************************
```

As m4 reads its input, it separates it into tokens. A token is either a name, a quoted string, or any single character, that is not a part of either a name or a string. Input to m4 can also contain comments.

```
                Names
                 Macro names

                Quoted strings
                 Quoting input to m4

                Other tokens
                 Other kinds of input tokens

                Comments
                 Comments in m4 input
```

1.9 m4.guide/Names

```
                Names
=====
```

A name is any sequence of letters, digits, and the character _ (underscore), where the first character is not a digit. If a name has a macro definition, it will be subject to macro expansion (see
Macros
).

Examples of legal names are: foo, _tmp, and name01.

1.10 m4.guide/Quoted strings

```
                Quoted strings
==============

   A quoted string is a sequence of characters surrounded by the quotes
` and ', where the number of start and end quotes within the string
balances.  The value of a string token is the text, with one level of
quotes stripped off.  Thus

     ``'

   is the empty string, and

     ``quoted''

   is the string

     `quoted'

   The quote characters can be changed at any time, using the builtin
macro changequote.  See
                Changequote
                 for more information.
```

1.11 m4.guide/Other tokens

```
Other tokens
============

   Any character, that is neither a part of a name, nor of a quoted
string, is a token by itself.
```

1.12 m4.guide/Comments

```
                Comments
========

   Comments in m4 are normally delimited by the characters # and
newline.  All characters between the comment delimiters are ignored,
but the entire comment (including the delimiters) is passed through to
the output--comments are not discarded by m4.

   Comments cannot be nested, so the first newline after a # ends the
comment.  The commenting effect of the begin comment character can be
inhibited by quoting it.

   The comment delimiters can be changed to any string at any time,
using the builtin macro changecom.  See
```

```
Changecom
 for more information.
```

1.13 m4.guide/Macros

```
                How to invoke macros
********************

   This chapter covers macro invocation, macro arguments and how macro
expansion is treated.



Invocation
 Macro invocation

Inhibiting Invocation
 Preventing macro invocation

Macro Arguments
 Macro arguments

Quoting Arguments
 On Quoting Arguments to macros

Macro expansion
 Expanding macros
```

1.14 m4.guide/Invocation

```
Macro invocation
================

   Macro invocations has one of the forms

     name

which is a macro invocation without any arguments, or

     name(arg1, arg2, ..., argN)

which is a macro invocation with N arguments.  Macros can have any
number of arguments.  All arguments are strings, but different macros
might interpret the arguments in different ways.

   The opening parenthesis must follow the NAME directly, with no
spaces in between.  If it does not, the macro is called with no
arguments at all.
```

For a macro call to have no arguments, the parentheses must be left out. The macro call

```
name()
```

is a macro call with one argument, which is the empty string, not a call with no arguments.

1.15 m4.guide/Inhibiting Invocation

```
                Preventing macro invocation
==========================
```

An innovation of the m4 language, compared to some of its predecessors (like Stratchey's GPM, for example), is the ability to recognize macro calls without resorting to any special, prefixed invocation character. While generally useful, this feature might sometimes be the source of spurious, unwanted macro calls. So, GNU m4 offers several mechanisms or techniques for inhibiting the recognition of names as macro calls.

First of all, many builtin macros cannot meaningfully be called without arguments. For any of these macros, whenever an opening parenthesis does not immediately follow their name, the builtin macro call is not triggered. This solves the most usual cases, like for include or eval. Later in this document, the sentence "This macro is recognized only when given arguments" refers to this specific provision.

There is also a command call option (--prefix-builtins, or -P) which requires all builtin macro names to be prefixed by m4_ for them to be recognized. The option has no effect whatsoever on user defined macros. For example, with this option, one has to write m4_dnl and even m4_m4exit.

If your version of GNU m4 has the changeword feature compiled in, there it offers far more flexibility in specifying the syntax of macro names, both builtin or user-defined. See

```
                Changeword
                 for more
```

information on this experimental feature.

Of course, the simplest way to prevent a name to be interpreted as a call to an existing macro is to quote it. The remainder of this section studies a little more deeply how quoting affects macro invocation, and how quoting can be used to inhibit macro invocation.

Even if quoting is usually done over the whole macro name, it can also be done over only a few characters of this name. It is also possible to quote the empty string, but this works only inside the name. For example:

```
‘divert’
```

```
     `d'ivert
     di`ver't
     div``'ert
```

all yield the string divert. While in both:

```
     ``'divert
     divert``'
```

the divert builtin macro will be called.

The output of macro evaluations is always rescanned. The following example would yield the string de, exactly as if m4 has been given substr(abcde, 3, 2) as input:

```
     define(`x', `substr(ab')
     define(`y', `cde, 3, 2)')
     x``'y
```

Unquoted strings on either side of a quoted string are subject to being recognized as macro names. In the following example, quoting the empty string allows for the dnl macro to be recognized as such:

```
     define(`macro', `di$1')
     macro(v)``'dnl
```

Without the quotes, this would rather yield the string divdnl followed by an end of line.

Quoting may prevent recognizing as a macro name the concatenation of a macro expansion with the surrounding characters. In this example:

```
     define(`macro', `di$1')
     macro(v)`ert'
```

the input will produce the string divert. If the quote was removed, the divert builtin would be called instead.

1.16 m4.guide/Macro Arguments

```
Macro arguments
===============
```

When a name is seen, and it has a macro definition, it will be expanded as a macro.

If the name is followed by an opening parenthesis, the arguments will be collected before the macro is called. If too few arguments are supplied, the missing arguments are taken to be the empty string. If there are too many arguments, the excess arguments are ignored.

Normally m4 will issue warnings if a builtin macro is called with an inappropriate number of arguments, but it can be suppressed with the -Q

command line option. For user defined macros, there is no check of the number of arguments given.

Macros are expanded normally during argument collection, and whatever commas, quotes and parentheses that might show up in the resulting expanded text will serve to define the arguments as well. Thus, if FOO expands to , b, c, the macro call

```
bar(a foo, d)
```

is a macro call with four arguments, which are a , b, c and d. To understand why the first argument contains whitespace, remember that leading unquoted whitespace is never part of an argument, but trailing whitespace always is.

1.17 m4.guide/Quoting Arguments

```
Quoting macro arguments
=======================
```

Each argument has leading unquoted whitespace removed. Within each argument, all unquoted parentheses must match. For example, if FOO is a macro,

```
foo(() (`(') `(')
```

is a macro call, with one argument, whose value is () (() (.

It is common practice to quote all arguments to macros, unless you are sure you want the arguments expanded. Thus, in the above example with the parentheses, the 'right' way to do it is like this:

```
foo(`() (() (')
```

It is, however, in certain cases necessary to leave out quotes for some arguments, and there is nothing wrong in doing it. It just makes life a bit harder, if you are not careful.

1.18 m4.guide/Macro expansion

```
Macro expansion
===============
```

When the arguments, if any, to a macro call have been collected, the macro is expanded, and the expansion text is pushed back onto the input (unquoted), and reread. The expansion text from one macro call might therefore result in more macros being called, if the calls are included, completely or partially, in the first macro calls' expansion.

Taking a very simple example, if FOO expands to bar, and BAR expands

to Hello world, the input

```
     foo
```

will expand first to bar, and when this is reread and expanded, into Hello world.

1.19 m4.guide/Definitions

```
                How to define new macros
************************
```

Macros can be defined, redefined and deleted in several different ways. Also, it is possible to redefine a macro, without losing a previous value, which can be brought back at a later time.

1.20 m4.guide/Define

```
                Defining a macro
================
```

The normal way to define or redefine macros is to use the builtin

```
define:

     define(NAME [, EXPANSION])

which defines NAME to expand to EXPANSION.  If EXPANSION is not given,
it is taken to be empty.

   The expansion of define is void.

   The following example defines the macro FOO to expand to the text
Hello World..

     define(‘foo’, ‘Hello world.’)
     =>
     foo
     =>Hello world.

   The empty line in the output is there because the newline is not a
part of the macro definition, and it is consequently copied to the
output.  This can be avoided by use of the macro dnl.  See
                Dnl
                , for
details.

   The macro define is recognized only with parameters.
```

1.21 m4.guide/Arguments

```
                Arguments to macros
==================

   Macros can have arguments.  The Nth argument is denoted by $n in the
expansion text, and is replaced by the Nth actual argument, when the
macro is expanded.  Here is a example of a macro with two arguments.
It simply exchanges the order of the two arguments.

     define(‘exch’, ‘$2, $1’)
     =>
     exch(arg1, arg2)
     =>arg2, arg1

   This can be used, for example, if you like the arguments to define
to be reversed.

     define(‘exch’, ‘$2, $1’)
     =>
     define(exch(``expansion text'', ``macro''))
     =>
     macro
     =>expansion text

   See
                Quoting Arguments
```

, for an explanation of the double quotes.

GNU m4 allows the number following the $ to consist of one or more digits, allowing macros to have any number of arguments. This is not so in UNIX implementations of m4, which only recognize one digit.

As a special case, the zero'th argument, $0, is always the name of the macro being expanded.

```
define(‘test’, ‘‘Macro name: $0’’)
=>
test
=>Macro name: test
```

If you want quoted text to appear as part of the expansion text, remember that quotes can be nested in quoted strings. Thus, in

```
define(‘foo’, ‘This is macro ‘foo’.’)
=>
foo
=>This is macro foo.
```

The foo in the expansion text is not expanded, since it is a quoted string, and not a name.

1.22 m4.guide/Pseudo Arguments

```
                Special arguments to macros
============================
```

There is a special notation for the number of actual arguments supplied, and for all the actual arguments.

The number of actual arguments in a macro call is denoted by $# in the expansion text. Thus, a macro to display the number of arguments given can be

```
define(‘nargs’, ‘$#’)
=>
nargs
=>0
nargs()
=>1
nargs(arg1, arg2, arg3)
=>3
```

The notation $* can be used in the expansion text to denote all the actual arguments, unquoted, with commas in between. For example

```
define(‘echo’, ‘$*’)
=>
echo(arg1,    arg2, arg3 , arg4)
=>arg1,arg2,arg3 ,arg4
```

Often each argument should be quoted, and the notation $@ handles that. It is just like $*, except that it quotes each argument. A simple example of that is:

```
define(‘echo’, ‘$@’)
=>
echo(arg1,    arg2, arg3 , arg4)
=>arg1,arg2,arg3 ,arg4
```

Where did the quotes go? Of course, they were eaten, when the expanded text were reread by m4. To show the difference, try

```
define(‘echo1’, ‘$*’)
=>
define(‘echo2’, ‘$@’)
=>
define(‘foo’, ‘This is macro ‘foo’.’)
=>
echo1(foo)
=>This is macro This is macro foo..
echo2(foo)
=>This is macro foo.
```

See
Trace
, if you do not understand this.

A $ sign in the expansion text, that is not followed by anything m4 understands, is simply copied to the macro expansion, as any other text is.

```
define(‘foo’, ‘$$$ hello $$$’)
=>
foo
=>$$$ hello $$$
```

If you want a macro to expand to something like $12, put a pair of quotes after the $. This will prevent m4 from interpreting the $ sign as a reference to an argument.

1.23 m4.guide/Undefine

Deleting a macro
==========

A macro definition can be removed with undefine:

```
undefine(NAME)
```

which removes the macro NAME. The macro name must necessarily be quoted, since it will be expanded otherwise.

The expansion of undefine is void.

```
foo
=>foo
define(‘foo’, ‘expansion text’)
=>
foo
=>expansion text
undefine(‘foo’)
=>
foo
=>foo
```

It is not an error for NAME to have no macro definition. In that case, undefine does nothing.

The macro undefine is recognized only with parameters.

1.24 m4.guide/Defn

```
Renaming macros
===============
```

It is possible to rename an already defined macro. To do this, you need the builtin defn:

```
defn(NAME)
```

which expands to the quoted definition of NAME. If the argument is not a defined macro, the expansion is void.

If NAME is a user-defined macro, the quoted definition is simply the quoted expansion text. If, instead, NAME is a builtin, the expansion is a special token, which points to the builtin’s internal definition. This token is only meaningful as the second argument to define (and pushdef), and is ignored in any other context.

Its normal use is best understood through an example, which shows how to rename undefine to zap:

```
define(‘zap’, defn(‘undefine’))
=>
zap(‘undefine’)
=>
undefine(‘zap’)
=>undefine(zap)
```

In this way, defn can be used to copy macro definitions, and also definitions of builtin macros. Even if the original macro is removed, the other name can still be used to access the definition.

The macro defn is recognized only with parameters.

1.25 m4.guide/Pushdef

```
Temporarily redefining macros
=============================

   It is possible to redefine a macro temporarily, reverting to the
previous definition at a later time.  This is done with the builtins
pushdef and popdef:

     pushdef(NAME [, EXPANSION])
     popdef(NAME)

which are quite analogous to define and undefine.

   These macros work in a stack-like fashion.  A macro is temporarily
redefined with pushdef, which replaces an existing definition of NAME,
while saving the previous definition, before the new one is installed.
If there is no previous definition, pushdef behaves exactly like define.

   If a macro has several definitions (of which only one is accessible),
the topmost definition can be removed with popdef.  If there is no
previous definition, popdef behaves like undefine.

     define(‘foo’, ‘Expansion one.’)
     =>
     foo
     =>Expansion one.
     pushdef(‘foo’, ‘Expansion two.’)
     =>
     foo
     =>Expansion two.
     popdef(‘foo’)
     =>
     foo
     =>Expansion one.
     popdef(‘foo’)
     =>
     foo
     =>foo

   If a macro with several definitions is redefined with define, the
topmost definition is replaced with the new definition.  If it is
removed with undefine, all the definitions are removed, and not only
the topmost one.

     define(‘foo’, ‘Expansion one.’)
     =>
     foo
     =>Expansion one.
     pushdef(‘foo’, ‘Expansion two.’)
     =>
     foo
     =>Expansion two.
     define(‘foo’, ‘Second expansion two.’)
     =>
     foo
```

```
     =>Second expansion two.
     undefine(‘foo’)
     =>
     foo
     =>foo
```

   It is possible to temporarily redefine a builtin with pushdef and defn.

   The macros pushdef and popdef are recognized only with parameters.

1.26 m4.guide/Indir

```
Indirect call of macros
=======================
```

   Any macro can be called indirectly with indir:

```
     indir(NAME, ...)
```

which results in a call to the macro NAME, which is passed the rest of the arguments. This can be used to call macros with "illegal" names (define allows such names to be defined):

```
     define(‘$$internal$macro’, ‘Internal macro (name ‘$0’)’)
     =>
     $$internal$macro
     =>$$internal$macro
     indir(‘$$internal$macro’)
     =>Internal macro (name $$internal$macro)
```

   The point is, here, that larger macro packages can have private macros defined, that will not be called by accident. They can only be called through the builtin indir.

1.27 m4.guide/Builtin

```
Indirect call of builtins
=========================
```

   Builtin macros can be called indirectly with builtin:

```
     builtin(NAME, ...)
```

which results in a call to the builtin NAME, which is passed the rest of the arguments. This can be used, if NAME has been given another definition that has covered the original.

   The macro builtin is recognized only with parameters.

1.28 m4.guide/Conditionals

```
                Conditionals, loops and recursion
*********************************
```

   Macros, expanding to plain text, perhaps with arguments, are not quite enough. We would like to have macros expand to different things, based on decisions taken at run-time. E.g., we need some kind of conditionals. Also, we would like to have some kind of loop construct, so we could do something a number of times, or while some condition is true.

1.29 m4.guide/Ifdef

```
Testing macro definitions
=========================
```

   There are two different builtin conditionals in m4. The first is ifdef:

```
     ifdef(NAME, STRING-1, opt STRING-2)
```

which makes it possible to test whether a macro is defined or not. If NAME is defined as a macro, ifdef expands to STRING-1, otherwise to STRING-2. If STRING-2 is omitted, it is taken to be the empty string (according to the normal rules).

```
     ifdef(`foo', ``foo' is defined', ``foo' is not defined')
     =>foo is not defined
     define(`foo', `')
     =>
     ifdef(`foo', ``foo' is defined', ``foo' is not defined')
     =>foo is defined
```

   The macro ifdef is recognized only with parameters.

1.30 m4.guide/Ifelse

```
Comparing strings
=================
```

   The other conditional, ifelse, is much more powerful. It can be used as a way to introduce a long comment, as an if-else construct, or as a multibranch, depending on the number of arguments supplied:

```
     ifelse(COMMENT)
     ifelse(STRING-1, STRING-2, EQUAL, opt NOT-EQUAL)
     ifelse(STRING-1, STRING-2, EQUAL, ...)
```

Used with only one argument, the ifelse simply discards it and produces no output. This is a common m4 idiom for introducing a block comment, as an alternative to repeatedly using dnl. This special usage is recognized by GNU m4, so that in this case, the warning about missing arguments is never triggered.

   If called with three or four arguments, ifelse expands into EQUAL, if STRING-1 and STRING-2 are equal (character for character), otherwise it expands to NOT-EQUAL.

```
     ifelse(foo, bar, 'true')
     =>
     ifelse(foo, foo, 'true')
     =>true
     ifelse(foo, bar, 'true', 'false')
     =>false
     ifelse(foo, foo, 'true', 'false')
     =>true
```

   However, ifelse can take more than four arguments. If given more than four arguments, ifelse works like a case or switch statement in traditional programming languages. If STRING-1 and STRING-2 are equal, ifelse expands into EQUAL, otherwise the procedure is repeated with the first three arguments discarded. This calls for an example:

```
     ifelse(foo, bar, 'third', gnu, gnats, 'sixth', 'seventh')
     =>seventh
```

   Naturally, the normal case will be slightly more advanced than these examples. A common use of ifelse is in macros implementing loops of various kinds.

   The macro ifelse is recognized only with parameters.

1.31 m4.guide/Loops

```
                Loops and recursion
==================
```

   There is no direct support for loops in m4, but macros can be recursive. There is no limit on the number of recursion levels, other than those enforced by your hardware and operating system.

Loops can be programmed using recursion and the conditionals described previously.

There is a builtin macro, shift, which can, among other things, be used for iterating through the actual arguments to a macro:

```
shift(...)
```

It takes any number of arguments, and expands to all but the first argument, separated by commas, with each argument quoted.

```
shift(bar)
=>
shift(foo, bar, baz)
=>bar,baz
```

An example of the use of shift is this macro, which reverses the order of its arguments:

```
define(‘reverse’, ‘ifelse($#, 0, , $#, 1, ‘‘$1’’,
       ‘reverse(shift($@)), ‘$1’’)’)
=>
reverse
=>
reverse(foo)
=>foo
reverse(foo, bar, gnats, and gnus)
=>and gnus, gnats, bar, foo
```

While not a very interesting macro, it does show how simple loops can be made with shift, ifelse and recursion.

Here is an example of a loop macro that implements a simple forloop. It can, for example, be used for simple counting:

```
forloop(‘i’, 1, 8, ‘i ’)
=>1 2 3 4 5 6 7 8
```

The arguments are a name for the iteration variable, the starting value, the final value, and the text to be expanded for each iteration. With this macro, the macro i is defined only within the loop. After the loop, it retains whatever value it might have had before.

For-loops can be nested, like

```
forloop(‘i’, 1, 4, ‘forloop(‘j’, 1, 8, ‘(i, j) ’)
’)
=>(1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8)
=>(2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8)
=>(3, 1) (3, 2) (3, 3) (3, 4) (3, 5) (3, 6) (3, 7) (3, 8)
=>(4, 1) (4, 2) (4, 3) (4, 4) (4, 5) (4, 6) (4, 7) (4, 8)
=>
```

The implementation of the forloop macro is fairly straightforward. The forloop macro itself is simply a wrapper, which saves the previous definition of the first argument, calls the internal macro _forloop,

```
and re-establishes the saved definition of the first argument.

   The macro _forloop expands the fourth argument once, and tests to
see if it is finished.  If it has not finished, it increments the
iteration variable (using the predefined macro incr, see
                Incr
                ), and
recurses.

   Here is the actual implementation of forloop:

     define(‘forloop’,
            ‘pushdef(‘$1’, ‘$2’)_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)popdef(‘$1’)’)
     define(‘_forloop’,
            ‘$4’’ifelse($1, ‘$3’, ,
           ‘define(‘$1’, incr($1))_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)’)’)

   Notice the careful use of quotes.  Only three macro arguments are
unquoted, each for its own reason.  Try to find out why these three
arguments are left unquoted, and see what happens if they are quoted.

   Now, even though these two macros are useful, they are still not
robust enough for general use. They lack even basic error handling of
cases like start value less than final value, and the first argument
not being a name.  Correcting these errors are left as an exercise to
the reader.
```

1.32 m4.guide/Debugging

```
                How to debug macros and input
*****************************

   When writing macros for m4, most of the time they woould not work as
intended (as is the case with most programming languages).  There is a
little support for macro debugging in m4.



                Dumpdef
                 Displaying macro definitions

                Trace
                 Tracing macro calls

                Debug Levels
                 Controlling debugging output

                Debug Output
                 Saving debugging output
```

1.33 m4.guide/Dumpdef

```
                Displaying macro definitions
============================
```

   If you want to see what a name expands into, you can use the builtin dumpdef:

```
     dumpdef(...)
```

which accepts any number of arguments. If called without any arguments, it displays the definitions of all known names, otherwise it displays the definitions of the names given. The output is printed directly on the standard error output.

   The expansion of dumpdef is void.

```
     define(‘foo’, ‘Hello world.’)
     =>
     dumpdef(‘foo’)
     error-->foo: ‘Hello world.’
     =>
     dumpdef(‘define’)
     error-->define:  <define>
     =>
```

   The last example shows how builtin macros definitions are displayed.

   See
               Debug Levels
                for information on controlling the details of the display.

1.34 m4.guide/Trace

```
                Tracing macro calls
==================
```

   It is possible to trace macro calls and expansions through the builtins traceon and traceoff:

```
     traceon(...)
     traceoff(...)
```

When called without any arguments, traceon and traceoff will turn tracing on and off, respectively, for all defined macros. When called with arguments, only the named macros are affected.

   The expansion of traceon and traceoff is void.

   Whenever a traced macro is called and the arguments have been collected, the call is displayed. If the expansion of the macro call

is not void, the expansion can be displayed after the call. The output is printed directly on the standard error output.

```
define(‘foo’, ‘Hello World.’)
=>
define(‘echo’, ‘$@’)
=>
traceon(‘foo’, ‘echo’)
=>
foo
error-->m4trace: -1- foo -> ‘Hello World.’
=>Hello World.
echo(gnus, and gnats)
error-->m4trace: -1- echo(‘gnus’, ‘and gnats’) -> ‘‘gnus’,‘and gnats’’
=>gnus,and gnats
```

The number between dashes is the depth of the expansion. It is one most of the time, signifying an expansion at the outermost level, but it increases when macro arguments contain unquoted macro calls.

See
Debug Levels
for information on controlling the details of the display.

1.35 m4.guide/Debug Levels

Controlling debugging output
==========

The -d option to m4 controls the amount of details presented, when using the macros described in the preceding sections.

The FLAGS following the option can be one or more of the following:

t
Trace all macro calls made in this invocation of m4.

a
Show the actual arguments in each macro call. This applies to all macro calls if the t flag is used, otherwise only the macros covered by calls of traceon.

e
Show the expansion of each macro call, if it is not void. This applies to all macro calls if the t flag is used, otherwise only the macros covered by calls of traceon.

q
Quote actual arguments and macro expansions in the display with the current quotes.

c

```
     Show several trace lines for each macro call.  A line is shown
     when the macro is seen, but before the arguments are collected; a
     second line when the arguments have been collected and a third
     line after the call has completed.

x
     Add a unique 'macro call id' to each line of the trace output.
     This is useful in connection with the c flag above.

f
     Show the name of the current input file in each trace output line.

l
     Show the the current input line number in each trace output line.

p
     Print a message when a named file is found through the path search
     mecanism (see
                Search Path
                ), giving the actual filename used.

i
     Print a message each time the current input file is changed,
     giving file name and input line number.

V
     A shorthand for all of the above flags.

   If no flags are specified with the -d option, the default is aeq.
The examples in the previous two sections assumed the default flags.

   There is a builtin macro debugmode, which allows on-the-fly control
of the debugging output format:

     debugmode(opt FLAGS)

The argument FLAGS should be a subset of the letters listed above.  As
special cases, if the argument starts with a +, the flags are added to
the current debug flags, and if it starts with a -, they are removed.
If no argument is present, the debugging flags are set to zero (as if
no -d was given), and with an empty argument the flags are reset to the
default.
```

1.36 m4.guide/Debug Output

```
Saving debugging output
=======================

   Debug and tracing output can be redirected to files using either the
-o option to m4, or with the builtin macro debugfile:

     debugfile(opt FILENAME)
```

will send all further debug and trace output to FILENAME. If FILENAME is empty, debug and trace output are discarded and if debugfile is called without any arguments, debug and trace output are sent to the standard error output.

1.37 m4.guide/Input Control

```
                Input control
*************
```

This chapter describes various builtin macros for controlling the input to m4.

1.38 m4.guide/Dnl

```
                Deleting whitespace in input
============================
```

The builtin dnl reads and discards all characters, up to and including the first newline:

```
dnl
```

and it is often used in connection with define, to remove the newline that follow the call to define. Thus

```
define(‘foo’, ‘Macro ‘foo’.’)dnl A very simple macro, indeed.
foo
=>Macro foo.
```

The input up to and including the next newline is discarded, as opposed to the way comments are treated (see

```
               Comments
               ).
```

   Usually, dnl is immediately followed by an end of line or some other whitespace. GNU m4 will produce a warning diagnostic if dnl is followed by an open parenthesis. In this case, dnl will collect and process all arguments, looking for a matching close parenthesis. All predictable side effects resulting from this collection will take place. dnl will return no output. The input following the matching close parenthesis up to and including the next newline, on whatever line containing it, will still be discarded.

1.39 m4.guide/Changequote

```
Changing the quote characters
=============================
```

   The default quote delimiters can be changed with the builtin changequote:

```
     changequote(opt START, opt END)
```

where START is the new start-quote delimiter and END is the new end-quote delimiter. If any of the arguments are missing, the default quotes (‘ and ’) are used instead of the void arguments.

   The expansion of changequote is void.

```
     changequote([, ])
     =>
     define([foo], [Macro [foo].])
     =>
     foo
     =>Macro foo.
```

   If no single character is appropriate, START and END can be of any length.

```
     changequote([[, ]])
     =>
     define([[foo]], [[Macro [[[foo]]].]])
     =>
     foo
     =>Macro [foo].
```

   Changing the quotes to the empty strings will effectively disable the quoting mechanism, leaving no way to quote text.

```
     define(‘foo’, ‘Macro ‘FOO’.’)
     =>
     changequote(, )
     =>
     foo
```

```
     =>Macro ‘FOO’.
     ‘foo’
     =>‘Macro ‘FOO’.’
```

   There is no way in m4 to quote a string containing an unmatched left quote, except using changequote to change the current quotes.

   Neither quote string should start with a letter or _ (underscore), as they will be confused with names in the input. Doing so disables the quoting mechanism.

1.40 m4.guide/Changecom

```
Changing comment delimiters
===========================
```

   The default comment delimiters can be changed with the builtin macro changecom:

```
     changecom(opt START, opt END)
```

where START is the new start-comment delimiter and END is the new end-comment delimiter. If any of the arguments are void, the default comment delimiters (# and newline) are used instead of the void arguments. The comment delimiters can be of any length.

   The expansion of changecom is void.

```
     define(‘comment’, ‘COMMENT’)
     =>
     # A normal comment
     =># A normal comment
     changecom(‘/*’, ‘*/’)
     =>
     # Not a comment anymore
     =># Not a COMMENT anymore
     But: /* this is a comment now */ while this is not a comment
     =>But: /* this is a comment now */ while this is not a COMMENT
```

   Note how comments are copied to the output, much as if they were quoted strings. If you want the text inside a comment expanded, quote the start comment delimiter.

   Calling changecom without any arguments disables the commenting mechanism completely.

```
     define(‘comment’, ‘COMMENT’)
     =>
     changecom
     =>
     # Not a comment anymore
     =># Not a COMMENT anymore
```

1.41 m4.guide/Changeword

```
Changing the lexical structure of words
=======================================

     The macro changeword and all associated functionnality is
     experimental.  It is only available if the --enable-changeword
     option was given to configure, at GNU m4 installation time.  The
     functionnality might change or even go away in the future.  Do not
     rely on it.  Please direct your comments about it the same way you
     would do for bugs.

   A file being processed by m4 is split into quoted strings, words
(potential macro names) and simple tokens (any other single character).
Initially a word is defined by the following regular expression:

     [_a-zA-Z][_a-zA-Z0-9]*

   Using changeword, you can change this regular expression.  Relaxing
m4's lexical rules might be useful (for example) if you wanted to
apply translations to a file of numbers:

     changeword('[_a-zA-Z0-9]+')
     define(1, 0)
     =>1

   Tightening the lexical rules is less useful, because it will
generally make some of the builtins unavailable.  You could use it to
prevent accidental call of builtins, for example:

     define('_indir', defn('indir'))
     changeword('_[_a-zA-Z0-9]*')
     esyscmd(foo)
     _indir('esyscmd', 'ls')

   Because m4 constructs its words a character at a time, there is a
restriction on the regular expressions that may be passed to
changeword.  This is that if your regular expression accepts
foo, it must also accept f and fo.

   changeword has another function.  If the regular expression supplied
contains any bracketed subexpressions, then text outside the first of
these is discarded before symbol lookup.  So:

     changecom('/*', '*/')
     changeword('#\([_a-zA-Z0-9]*\)')
     #esyscmd(ls)

   m4 now requires a # mark at the beginning of every macro invocation,
so one can use m4 to preprocess shell scripts without getting shift
commands swallowed, and plain text without losing various common words.

   m4's macro substitution is based on text, while TeX's is based on
tokens.  changeword can throw this difference into relief.  For
example, here is the same idea represented in TeX and m4.  First, the
TeX version:
```

```
     \def\a{\message{Hello}}
     \catcode`@=0
     \catcode`\=12
     =>@a
     =>@bye
```

Then, the m4 version:

```
     define(a, `errprint(`Hello')')
     changeword(`@\([_a-zA-Z0-9]*\)')
     =>@a
```

In the TeX example, the first line defines a macro a to print the message Hello. The second line defines @ to be usable instead of \ as an escape character. The third line defines \ to be a normal printing character, not an escape. The fourth line invokes the macro a. So, when TeX is run on this file, it displays the message Hello.

When the m4 example is passed through m4, it outputs errprint(Hello). The reason for this is that TeX does lexical analysis of macro definition when the macro is defined. m4 just stores the text, postponing the lexical analysis until the macro is used.

You should note that using changeword will slow m4 down by a factor of about seven.

1.42 m4.guide/M4wrap

```
Saving input
============
```

It is possible to `save' some text until the end of the normal input has been seen. Text can be saved, to be read again by m4 when the normal input has been exhausted. This feature is normally used to initiate cleanup actions before normal exit, e.g., deleting temporary files.

To save input text, use the builtin m4wrap:

```
  m4wrap(STRING, ...)
```

which stores STRING and the rest of the arguments in a safe place, to be reread when end of input is reached.

```
     define(`cleanup', `This is the `cleanup' actions.
     ')
     =>
     m4wrap(`cleanup')
     =>
     This is the first and last normal input line.
     =>This is the first and last normal input line.
     ^D
     =>This is the cleanup actions.
```

The saved input is only reread when the end of normal input is seen, and not if m4exit is used to exit m4.

It is safe to call m4wrap from saved text, but then the order in which the saved text is reread is undefined. If m4wrap is not used recursively, the saved pieces of text are reread in the opposite order in which they were saved (LIFO--last in, first out).

1.43 m4.guide/File Inclusion

```
                File inclusion
**************
```

m4 allows you to include named files at any point in the input.

1.44 m4.guide/Include

```
Including named files
=====================
```

There are two builtin macros in m4 for including files:

```
include(FILENAME)
sinclude(FILENAME)
```

both of which cause the file named FILENAME to be read by m4. When the end of the file is reached, input is resumed from the previous input file.

The expansion of include and sinclude is therefore the contents of FILENAME.

It is an error for an included file not to exist. If you do not want error messages about non-existent files, sinclude can be used to include a file, if it exists, expanding to nothing if it does not.

```
include(‘no-such-file’)
=>
error-->30.include:2: m4: Cannot open no-such-file: No such file or directory
sinclude(‘no-such-file’)
```

```
     =>

   Assume in the following that the file incl.m4 contains the lines:
     Include file start
     foo
     Include file end

Normally file inclusion is used to insert the contents of a file into
the input stream.  The contents of the file will be read by m4 and
macro calls in the file will be expanded:

     define(‘foo’, ‘FOO’)
     =>
     include(‘incl.m4’)
     =>Include file start
     =>FOO
     =>Include file end
     =>

   The fact that include and sinclude expand to the contents of the
file can be used to define macros that operate on entire files.  Here
is an example, which defines bar to expand to the contents of incl.m4:

     define(‘bar’, include(‘incl.m4’))
     =>
     This is ‘bar’:  >>>bar<<<
     =>This is bar:  >>>Include file start
     =>foo
     =>Include file end
     =><<<

   This use of include is not trivial, though, as files can contain
quotes, commas and parentheses, which can interfere with the way the m4
parser works.

   The builtin macros include and sinclude are recognized only when
given arguments.
```

1.45 m4.guide/Search Path

```
                Searching for include files
============================

   GNU m4 allows included files to be found in other directories than
the current working directory.

   If a file is not found in the current working directory, and the file
name is not absolute, the file will be looked for in a specified search
path.  First, the directories specified with the -I option will be
searched, in the order found on the command line.  Second, if the
M4PATH environment variable is set, it is expected to contain a
colon-separated list of directories, which will be searched in order.

   If the automatic search for include-files causes trouble, the p
```

```
debug flag (see
                Debug Levels
                ) can help isolate the problem.
```

1.46 m4.guide/Diversions

```
                Diverting and undiverting output
*********************************
```

Diversions are a way of temporarily saving output. The output of m4 can at any time be diverted to a temporary file, and be reinserted into the output stream, undiverted, again at a later time.

Numbered diversions are counted from 0 upwards, diversion number 0 being the normal output stream. The number of simultaneous diversions is limited mainly by the memory used to describe them, because GNU m4 tries to keep diversions in memory. However, there is a limit to the overall memory usable by all diversions taken altogether (512K, currently). When this maximum is about to be exceeded, a temporary file is opened to receive the contents of the biggest diversion still in memory, freeing this memory for other diversions. So, it is theoretically possible that the number of diversions be limited by the number of available file descriptors.

```
Divert
 Diverting output

Undivert
 Undiverting output

Divnum
 Diversion numbers

Cleardiv
 Discarding diverted text
```

1.47 m4.guide/Divert

```
Diverting output
================
```

Output is diverted using divert:

```
  divert(opt NUMBER)
```

where NUMBER is the diversion to be used. If NUMBER is left out, it is

assumed to be zero.

   The expansion of divert is void.

   When all the m4 input will have been processed, all existing diversions are automatically undiverted, in numerical order.

```
divert(1)
This text is diverted.
divert
=>
This text is not diverted.
=>This text is not diverted.
^D
=>
=>This text is diverted.
```

   Several calls of divert with the same argument do not overwrite the previous diverted text, but append to it.

   If output is diverted to a non-existent diversion, it is simply discarded. This can be used to suppress unwanted output. A common example of unwanted output is the trailing newlines after macro definitions. Here is how to avoid them.

```
divert(-1)
define(‘foo’, ‘Macro ‘foo’.’)
define(‘bar’, ‘Macro ‘bar’.’)
divert
=>
```

   This is a common programming idiom in m4.

1.48 m4.guide/Undivert

```
                Undiverting output
==================
```

   Diverted text can be undiverted explicitly using the builtin undivert:

```
undivert(opt NUMBER, ...)
```

which undiverts the diversions given by the arguments, in the order given. If no arguments are supplied, all diversions are undiverted, in numerical order.

   The expansion of undivert is void.

```
divert(1)
This text is diverted.
divert
=>
This text is not diverted.
```

```
     =>This text is not diverted.
     undivert(1)
     =>
     =>This text is diverted.
     =>
```

   Notice the last two blank lines. One of them comes from the newline following undivert, the other from the newline that followed the divert! A diversion often starts with a blank line like this.

   When diverted text is undiverted, it is not reread by m4, but rather copied directly to the current output, and it is therefore not an error to undivert into a diversion.

   When a diversion has been undiverted, the diverted text is discarded, and it is not possible to bring back diverted text more than once.

```
     divert(1)
     This text is diverted first.
     divert(0)undivert(1)dnl
     =>
     =>This text is diverted first.
     undivert(1)
     =>
     divert(1)
     This text is also diverted but not appended.
     divert(0)undivert(1)dnl
     =>
     =>This text is also diverted but not appended.
```

   Attempts to undivert the current diversion are silently ignored.

   GNU m4 allows named files to be undiverted. Given a non-numeric argument, the contents of the file named will be copied, uninterpreted, to the current output. This complements the builtin include (see

               Include
). To illustrate the difference, assume the file foo contains the word bar:

```
     define(‘bar’, ‘BAR’)
     =>
     undivert(‘foo’)
     =>bar
     =>
     include(‘foo’)
     =>BAR
     =>
```

1.49 m4.guide/Divnum

```
Diversion numbers
=================
```

```
   The builtin divnum:

     divnum

expands to the number of the current diversion.

     Initial divnum
     =>Initial 0
     divert(1)
     Diversion one: divnum
     divert(2)
     Diversion two: divnum
     divert
     =>
     ^D
     =>
     =>Diversion one: 1
     =>
     =>Diversion two: 2

   The last call of divert without argument is necessary, since the
undiverted text would otherwise be diverted itself.
```

1.50 m4.guide/Cleardiv

```
Discarding diverted text
========================

   Often it is not known, when output is diverted, whether the diverted
text is actually needed.  Since all non-empty diversion are brought back
on the main output stream when the end of input is seen, a method of
discarding a diversion is needed.  If all diversions should be
discarded, the easiest is to end the input to m4 with divert(-1)
followed by an explicit undivert:

     divert(1)
     Diversion one: divnum
     divert(2)
     Diversion two: divnum
     divert(-1)
     undivert
     ^D

No output is produced at all.

   Clearing selected diversions can be done with the following macro:

     define(`cleardivert',
     `pushdef(`_num', divnum)divert(-1)undivert($@)divert(_num)popdef(`_num')')
     =>

   It is called just like undivert, but the effect is to clear the
diversions, given by the arguments.  (This macro has a nasty bug!  You
```

should try to see if you can find it and correct it.)

1.51 m4.guide/Text handling

```
                Macros for text handling
************************

   There are a number of builtins in m4 for manipulating text in
various ways, extracting substrings, searching, substituting, and so on.



                Len
                 Calculating length of strings

                Index
                 Searching for substrings

                Regexp
                 Searching for regular expressions

                Substr
                 Extracting substrings

                Translit
                 Translating characters

                Patsubst
                 Substituting text by regular expression

                Format
                 Formatting strings (printf-like)
```

1.52 m4.guide/Len

```
Calculating length of strings
=============================

   The length of a string can be calculated by len:

     len(STRING)

which expands to the length of STRING, as a decimal number.

     len()
     =>0
     len(‘abcdef’)
     =>6
```

The builtin macro len is recognized only when given arguments.

1.53 m4.guide/Index

```
Searching for substrings
========================
```

Searching for substrings is done with index:

```
index(STRING, SUBSTRING)
```

which expands to the index of the first occurrence of SUBSTRING in STRING. The first character in STRING has index 0. If SUBSTRING does not occur in STRING, index expands to -1.

```
index(‘gnus, gnats, and armadillos’, ‘nat’)
=>7
index(‘gnus, gnats, and armadillos’, ‘dag’)
=>-1
```

The builtin macro index is recognized only when given arguments.

1.54 m4.guide/Regexp

```
Searching for regular expressions
=================================
```

Searching for regular expressions is done with the builtin regexp:

```
regexp(STRING, REGEXP, opt REPLACEMENT)
```

which searches for REGEXP in STRING. The syntax for regular expressions is the same as in GNU Emacs. See Syntax of Regular Expressions.

If REPLACEMENT is omitted, regexp expands to the index of the first match of REGEXP in STRING. If REGEXP does not match anywhere in STRING, it expands to -1.

```
regexp(‘GNUs not Unix’, ‘\<[a-z]\w+’)
=>5
regexp(‘GNUs not Unix’, ‘\<Q\w*’)
=>-1
```

If REPLACEMENT is supplied, regexp changes the expansion to this argument, with \N substituted by the text matched by the Nth parenthesized sub-expression of REGEXP, \& being the text the entire regular expression matched.

```
regexp(‘GNUs not Unix’, ‘\w\(\w+\)$’, ‘*** \& *** \1 ***’)
```

```
     =>*** Unix *** nix ***

   The builtin macro regexp is recognized only when given arguments.
```

1.55 m4.guide/Substr

```
Extracting substrings
=====================

   Substrings are extracted with substr:

     substr(STRING, FROM, opt LENGTH)

which expands to the substring of STRING, which starts at index FROM,
and extends for LENGTH characters, or to the end of STRING, if LENGTH
is omitted.  The starting index of a string is always 0.

     substr(‘gnus, gnats, and armadillos’, 6)
     =>gnats, and armadillos
     substr(‘gnus, gnats, and armadillos’, 6, 5)
     =>gnats

   The builtin macro substr is recognized only when given arguments.
```

1.56 m4.guide/Translit

```
Translating characters
======================

   Character translation is done with translit:

     translit(STRING, CHARS, REPLACEMENT)

which expands to STRING, with each character that occurs in CHARS
translated into the character from REPLACEMENT with the same index.

   If REPLACEMENT is shorter than CHARS, the excess characters are
deleted from the expansion.  If REPLACEMENT is omitted, all characters
in STRING, that are present in CHARS are deleted from the expansion.

   Both CHARS and REPLACEMENT can contain character-ranges, e.g., a-z
(meaning all lowercase letters) or 0-9 (meaning all digits).  To
include a dash - in CHARS or REPLACEMENT, place it first or last.

   It is not an error for the last character in the range to be ‘larger’
than the first.  In that case, the range runs backwards, i.e., 9-0
means the string 9876543210.

     translit(‘GNUs not Unix’, ‘A-Z’)
     =>s not nix
```

```
translit(‘GNUs not Unix’, ‘a-z’, ‘A-Z’)
=>GNUS NOT UNIX
translit(‘GNUs not Unix’, ‘A-Z’, ‘z-a’)
=>tmfs not fnix
```

The first example deletes all uppercase letters, the second converts lowercase to uppercase, and the third ‘mirrors’ all uppercase letters, while converting them to lowercase. The two first cases are by far the most common.

The builtin macro translit is recognized only when given arguments.

1.57 m4.guide/Patsubst

```
Substituting text by regular expression
=======================================
```

Global substitution in a string is done by patsubst:

```
patsubst(STRING, REGEXP, opt REPLACEMENT)
```

which searches STRING for matches of REGEXP, and substitutes REPLACEMENT for each match. The syntax for regular expressions is the same as in GNU Emacs.

The parts of STRING that are not covered by any match of REGEXP are copied to the expansion. Whenever a match is found, the search proceeds from the end of the match, so a character from STRING will never be substituted twice. If REGEXP matches a string of zero length, the start position for the search is incremented, to avoid infinite loops.

When a replacement is to be made, REPLACEMENT is inserted into the expansion, with \N substituted by the text matched by the Nth parenthesized sub-expression of REGEXP, \& being the text the entire regular expression matched.

The REPLACEMENT argument can be omitted, in which case the text matched by REGEXP is deleted.

```
patsubst(‘GNUs not Unix’, ‘^’, ‘OBS: ’)
=>OBS: GNUs not Unix
patsubst(‘GNUs not Unix’, ‘\<’, ‘OBS: ’)
=>OBS: GNUs OBS: not OBS: Unix
patsubst(‘GNUs not Unix’, ‘\w*’, ‘(\&)’)
=>(GNUs)() (not)() (Unix)
patsubst(‘GNUs not Unix’, ‘\w+’, ‘(\&)’)
=>(GNUs) (not) (Unix)
patsubst(‘GNUs not Unix’, ‘[A-Z][a-z]+’)
=>GN not
```

Here is a slightly more realistic example, which capitalizes individual word or whole sentences, by substituting calls of the macros upcase and downcase into the strings.

```
     define(‘upcase’, ‘translit(‘$*’, ‘a-z’, ‘A-Z’)’)dnl
     define(‘downcase’, ‘translit(‘$*’, ‘A-Z’, ‘a-z’)’)dnl
     define(‘capitalize1’,
          ‘regexp(‘$1’, ‘^\(\w\)\(\w*\)’, ‘upcase(‘\1’)’’downcase(‘\2’)’)’)dnl
     define(‘capitalize’,
          ‘patsubst(‘$1’, ‘\w+’, ‘capitalize1(‘\&’)’)’)dnl
     capitalize(‘GNUs not Unix’)
     =>Gnus Not Unix
```

The builtin macro patsubst is recognized only when given arguments.

1.58 m4.guide/Format

```
                Formatted output
================
```

Formatted output can be made with format:

```
     format(FORMAT-STRING, ...)
```

which works much like the C function printf. The first argument is a format string, which can contain % specifications, and the expansion of format is the formatted string.

Its use is best described by a few examples:

```
     define(‘foo’, ‘The brown fox jumped over the lazy dog’)
     =>
     format(‘The string "%s" is %d characters long’, foo, len(foo))
     =>The string "The brown fox jumped over the lazy dog" is 38 characters long
```

Using the forloop macro defined in See
Loops
, this example shows how format can be used to produce tabular output.

```
     forloop(‘i’, 1, 10, ‘format(‘%6d squared is %10d
     ’, i, eval(i**2))’)
     =>     1 squared is      1
     =>     2 squared is      4
     =>     3 squared is      9
     =>     4 squared is     16
     =>     5 squared is     25
     =>     6 squared is     36
     =>     7 squared is     49
     =>     8 squared is     64
     =>     9 squared is     81
     =>    10 squared is    100
```

The builtin format is modeled after the ANSI C printf function, and supports the normal % specifiers: c, s, d, o, x, X, u, e, E and f; it supports field widths and precisions, and the modifiers +, -, , 0, #, h and l. For more details on the functioning of printf, see

the C Library Manual.

1.59 m4.guide/Arithmetic

```
                Macros for doing arithmetic
**************************
```

Integer arithmetic is included in m4, with a C-like syntax. As convenient shorthands, there are builtins for simple increment and decrement operations.

1.60 m4.guide/Incr

```
Decrement and increment operators
=================================
```

Increment and decrement of integers are supported using the builtins incr and decr:

```
     incr(NUMBER)
     decr(NUMBER)
```

which expand to the numerical value of NUMBER, incremented, or decremented, respectively, by one.

```
     incr(4)
     =>5
     decr(7)
     =>6
```

The builtin macros incr and decr are recognized only when given arguments.

1.61 m4.guide/Eval

Evaluating integer expressions

Integer expressions are evaluated with eval:

```
eval(EXPRESSION, opt RADIX, opt WIDTH)
```

which expands to the value of EXPRESSION.

Expressions can contain the following operators, listed in order of decreasing precedence.

`-`
: Unary minus

`**`
: Exponentiation

`*  /  %`
: Multiplication, division and modulo

`+  -`
: Addition and subtraction

`<<  >>`
: Shift left or right

`==  !=  >  >=  <  <=`
: Relational operators

`!`
: Logical negation

`~`
: Bitwise negation

`&`
: Bitwise and

`^`
: Bitwise exclusive-or

`|`
: Bitwise or

`&&`
: Logical and

`||`
: Logical or

All operators, except exponentiation, are left associative.

Note that many m4 implementations use ^ as an alternate operator for the exponentiation, while many others use ^ for the bitwise exclusive-or. GNU m4 changed its behavior: it used to exponentiate for

^, it now computes the bitwise exclusive-or.

Numbers without special prefix are given decimal. A simple 0 prefix introduces an octal number. 0x introduces an hexadecimal number. 0b introduces a binary number. 0r introduces a number expressed in any radix between 1 and 36: the prefix should be immediately followed by the decimal expression of the radix, a colon, then the digits making the number. For any radix, the digits are 0, 1, 2, ... Beyond 9, the digits are a, b ... up to z. Lower and upper case letters can be used interchangeably in numbers prefixes and as number digits.

Parentheses may be used to group subexpressions whenever needed. For the relational operators, a true relation returns 1, and a false relation return 0.

Here are a few examples of use of eval.

```
eval(-3 * 5)
=>-15
eval(index(‘Hello world’, ‘llo’) >= 0)
=>1
define(‘square’, ‘eval(($1)**2)’)
=>
square(9)
=>81
square(square(5)+1)
=>676
define(‘foo’, ‘666’)
=>
eval(‘foo’/6)
error-->51.eval:14: m4: Bad expression in eval: foo/6
=>
eval(foo/6)
=>111
```

As the second to last example shows, eval does not handle macro names, even if they expand to a valid expression (or part of a valid expression). Therefore all macros must be expanded before they are passed to eval.

If RADIX is specified, it specifies the radix to be used in the expansion. The default radix is 10. The result of eval is always taken to be signed. The WIDTH argument specifies a minimum output width. The result is zero-padded to extend the expansion to the requested width.

```
eval(666, 10)
=>666
eval(666, 11)
=>556
eval(666, 6)
=>3030
eval(666, 6, 10)
=>0000003030
eval(-666, 6, 10)
=>-000003030
```

Take note that RADIX cannot be larger than 36.

The builtin macro eval is recognized only when given arguments.

1.62 m4.guide/UNIX commands

```
                Running UNIX commands
**********************
```

There are a few builtin macros in m4 that allow you to run UNIX commands from within m4.

Syscmd
 Executing simple commands

Esyscmd
 Reading the output of commands

Sysval
 Exit codes

Maketemp
 Making names for temporary files

1.63 m4.guide/Syscmd

```
                Executing simple commands
=========================
```

Any shell command can be executed, using syscmd:

```
  syscmd(SHELL-COMMAND)
```

which executes SHELL-COMMAND as a shell command.

The expansion of syscmd is void, not the output from SHELL-COMMAND! Output or error messages from SHELL-COMMAND are not read by m4. See

Esyscmd
 if you need to process the command output.

Prior to executing the command, m4 flushes its output buffers. The default standard input, output and error of SHELL-COMMAND are the same as those of m4.

The builtin macro syscmd is recognized only when given arguments.

1.64 m4.guide/Esyscmd

```
Reading the output of commands
==============================

   If you want m4 to read the output of a UNIX command, use esyscmd:

     esyscmd(SHELL-COMMAND)

which expands to the standard output of the shell command SHELL-COMMAND.

   Prior to executing the command, m4 flushes its output buffers.  The
default standard input and error output of SHELL-COMMAND are the same
as those of m4.  The error output of SHELL-COMMAND is not a part of the
expansion: it will appear along with the error output of m4.

   Assume you are positioned into the checks directory of GNU m4
distribution, then:

     define(‘vice’, ‘esyscmd(grep Vice ../COPYING)’)
     =>
     vice
     =>  Ty Coon, President of Vice
     =>

   Note how the expansion of esyscmd has a trailing newline.

   The builtin macro esyscmd is recognized only when given arguments.
```

1.65 m4.guide/Sysval

```
Exit codes
==========

   To see whether a shell command succeeded, use sysval:

     sysval

which expands to the exit status of the last shell command run with
syscmd or esyscmd.

     syscmd(‘false’)
     =>
     ifelse(sysval, 0, zero, non-zero)
     =>non-zero
     syscmd(‘true’)
     =>
     sysval
     =>0
```

1.66 m4.guide/Maketemp

```
Making names for temporary files
================================
```

   Commands specified to syscmd or esyscmd might need a temporary file, for output or for some other purpose. There is a builtin macro, maketemp, for making temporary file names:

```
     maketemp(TEMPLATE)
```

which expands to a name of a non-existent file, made from the string TEMPLATE, which should end with the string XXXXXX. The six X's are then replaced, usually with something that includes the process id of the m4 process, in order to make the filename unique.

```
     maketemp(`/tmp/fooXXXXXX')
     =>/tmp/fooa07346
     maketemp(`/tmp/fooXXXXXX')
     =>/tmp/fooa07346
```

   As seen in the example, several calls of maketemp might expand to the same string, since the selection criteria is whether the file exists or not. If a file has not been created before the next call, the two macro calls might expand to the same name.

   The builtin macro maketemp is recognized only when given arguments.

1.67 m4.guide/Miscellaneous

```
                Miscellaneous builtin macros
****************************
```

   This chapter describes various builtins, that do not really belong in any of the previous chapters.

```
               Errprint
                Printing error messages

               M4exit
                Exiting from m4
```

1.68 m4.guide/Errprint

```
Printing error messages
=======================

   You can print error messages using errprint:

     errprint(MESSAGE, ...)

which simply prints MESSAGE and the rest of the arguments on the
standard error output.

   The expansion of errprint is void.

     errprint(`Illegal arguments to forloop
     ')
     error-->Illegal arguments to forloop
     =>

   A trailing newline is not printed automatically, so it must be
supplied as part of the argument, as in the example.  (BSD flavored
m4's do append a trailing newline on each errprint call).

   To make it possible to specify the location of the error, two
utility builtins exist:

     __file__
     __line__

which expands to the quoted name of the current input file, and the
current input line number in that file.

     errprint(`m4:'__file__:__line__: `Input error
     ')
     error-->m4:56.errprint:2: Input error
     =>
```

1.69 m4.guide/M4exit

```
                Exiting from m4
===============

   If you need to exit from m4 before the entire input has been read,
you can use m4exit:

     m4exit(opt CODE)

which causes m4 to exit, with exit code CODE.  If CODE is left out, the
exit code is zero.

     define(`fatal_error', `errprint(`m4: '__file__: __line__`: fatal error: $*
     ')m4exit(1)')
     =>
     fatal_error(`This is a BAD one, buster')
     error-->m4: 57.m4exit: 5: fatal error: This is a BAD one, buster
```

After this macro call, m4 will exit with exit code 1. This macro is only intended for error exits, since the normal exit procedures are not followed, e.g., diverted text is not undiverted, and saved text (see

M4wrap
) is not reread.

1.70 m4.guide/Frozen files

Fast loading of frozen states
****

Some bigger m4 applications may be built over a common base containing hundreds of definitions and other costly initializations. Usually, the common base is kept in one or more declarative files, which files are listed on each m4 invocation prior to the user's input file, or else, include'd from this input file.

Reading the common base of a big application, over and over again, may be time consuming. GNU m4 offers some machinery to speed up the start of an application using lengthy common bases. Presume the user repeatedly uses:

```
m4 base.m4 input.m4
```

with a varying contents of input.m4, but a rather fixed contents for base.m4. Then, the user might rather execute:

```
m4 -F base.m4f base.m4
```

once, and further execute, as often as needed:

```
m4 -R base.m4f input.m4
```

with the varying input. The first call, containing the -F option, only reads and executes file base.m4, so defining various application macros and computing other initializations. Only once the input file base.m4 has been completely processed, GNU m4 produces on base.m4f a frozen file, that is, a file which contains a kind of snapshot of the m4 internal state.

Later calls, containing the -R option, are able to reload the internal state of m4's memory, from base.m4f, prior to reading any other input files. By this mean, instead of starting with a virgin copy of m4, input will be read after having effectively recovered the effect of a prior run. In our example, the effect is the same as if file base.m4 has been read anew. However, this effect is achieved a lot faster.

Only one frozen file may be created or read in any one m4 invocation. It is not possible to recover two frozen files at once. However, frozen files may be updated incrementally, through using -R

and -F options simultaneously. For example, if some care is taken, the command:

```
m4 file1.m4 file2.m4 file3.m4 file4.m4
```

could be broken down in the following sequence, accumulating the same output:

```
m4 -F file1.m4f file1.m4
m4 -R file1.m4f -F file2.m4f file2.m4
m4 -R file2.m4f -F file3.m4f file3.m4
m4 -R file3.m4f file4.m4
```

Some care is necessary because not every effort has been made for this to work in all cases. In particular, the trace attribute of macros is not handled, nor the current setting of changeword. Also, interactions for some options of m4 being used in one call and not for the next, have not been fully analyzed yet. On the other end, you may be confident that stacks of pushdef'ed definitions are handled correctly, so are undefine'd or renamed builtins, changed strings for quotes or comments.

When an m4 run is to be frozen, the automatic undiversion which takes place at end of execution is inhibited. Instead, all positively numbered diversions are saved into the frozen file. The active diversion number is also transmitted.

A frozen file to be reloaded need not reside in the current directory. It is looked up the same way as an include file (see

Search Path
).

Frozen files are sharable across architectures. It is safe to write a frozen file one one machine and read it on another, given that the second machine uses the same, or a newer version of GNU m4. These are simple (editable) text files, made up of directives, each starting with a capital letter and ending with a newline (NL). Wherever a directive is expected, the character # introduces a comment line, empty lines are also ignored. In the following descriptions, LENGTHs always refer to corresponding STRINGs. Numbers are always expressed in decimal. The directives are:

V NUMBER NL
: Confirms the format of the file. NUMBER should be 1.

C LENGTH1 , LENGTH2 NL STRING1 STRING2 NL
: Uses STRING1 and STRING2 as the beginning comment and end comment strings.

Q LENGTH1 , LENGTH2 NL STRING1 STRING2 NL
: Uses STRING1 and STRING2 as the beginning quote and end quote strings.

F LENGTH1 , LENGTH2 NL STRING1 STRING2 NL
: Defines, through pushdef, a definition for STRING1 expanding to the function whose builtin name is STRING2.

```
T LENGTH1 , LENGTH2 NL STRING1 STRING2 NL
     Defines, though pushdef, a definition for STRING1 expanding to the
     text given by STRING2.

D NUMBER, LENGTH NL STRING NL
     Selects diversion NUMBER, making it current, then copy STRING in
     the current diversion.  NUMBER may be a negative number for a
     non-existing diversion.  To merely specify an active selection,
     use this command with an empty STRING.  With 0 as the diversion
     NUMBER, STRING will be issued on standard output at reload time,
     however this may not be produced from within m4.
```

1.71 m4.guide/Compatibility

```
                Compatibility with other versions of m4
**************************************
```

This chapter describes the differences between this implementation of m4, and the implementation found under UNIX, notably System V, Release 3.

There are also differences in BSD flavors of m4. No attempt is made to summarize these here.

1.72 m4.guide/Extensions

```
                Extensions in GNU m4
====================
```

This version of m4 contains a few facilities, that do not exist in System V m4. These extra facilities are all suppressed by using the -G command line option, unless overridden by other command line options.

* In the $ N notation for macro arguments, N can contain several digits, while the System V m4 only accepts one digit. This allows macros in GNU m4 to take any number of arguments, and not only

nine (see
Arguments
).

* Files included with include and sinclude are sought in a user specified search path, if they are not found in the working directory. The search path is specified by the -I option and the M4PATH environment variable (see
Search Path
).

* Arguments to undivert can be non-numeric, in which case the named file will be included uninterpreted in the output (see
Undivert
).

* Formatted output is supported through the format builtin, which is modeled after the C library function printf (see
Format
).

* Searches and text substitution through regular expressions are supported by the regexp (see
Regexp
) and patsubst (see
Patsubst
)
builtins.

* The output of shell commands can be read into m4 with esyscmd (see

Esyscmd
).

* There is indirect access to any builtin macro with builtin (see

Builtin
).

* Macros can be called indirectly through indir (see
Indir
).

* The name of the current input file and the current input line number are accessible through the builtins __file__ and __line__ (see
Errprint
).

* The format of the output from dumpdef and macro tracing can be controlled with debugmode (see
Debug Levels
).

* The destination of trace and debug output can be controlled with debugfile (see
Debug Output

).

In addition to the above extensions, GNU m4 implements the following command line options: -F, -G, -I, -L, -R, -V, -W, -d, -l, -o and -t. See

Invoking m4
, for a description of these options.

Also, the debugging and tracing facilities in GNU m4 are much more extensive than in most other versions of m4.

1.73 m4.guide/Incompatibilities

Facilities in System V m4 not in GNU m4
==========

The version of m4 from System V contains a few facilities that have not been implemented in GNU m4 yet.

* System V m4 supports multiple arguments to defn. This is not implemented in GNU m4. Its usefulness is unclear to me.

1.74 m4.guide/Other Incompat

Other incompatibilities
==========

There are a few other incompatibilities between this implementation of m4, and the System V version.

* GNU m4 implements sync lines differently from System V m4, when text is being diverted. GNU m4 outputs the sync lines when the text is being diverted, and System V m4 when the diverted text is being brought back.

  The problem is which lines and filenames should be attached to text that is being, or has been, diverted. System V m4 regards all the diverted text as being generated by the source line containing the undivert call, whereas GNU m4 regards the diverted text as being generated at the time it is diverted.

  I expect the sync line option to be used mostly when using m4 as a front end to a compiler. If a diverted line causes a compiler error, the error messages should most probably refer to the place where the diversion were made, and not where it was inserted again.

* GNU m4 makes no attempt at prohiting autoreferential definitions like:

```
     define(‘x’, ‘x’)
     define(‘x’, ‘x ’)
```

There is nothing inherently wrong with defining x to return x. The wrong thing is to expand x unquoted. In m4, one might use macros to hold strings, as we do for variables in other programming languages, further checking them with:

```
     ifelse(defn(‘HOLDER’), ‘VALUE’, ...)
```

In cases like this one, an interdiction for a macro to hold its own name would be a useless limitation. Of course, this leave more rope for the GNU m4 user to hang himself! Rescanning hangs may be avoided through careful programming, a little like for endless loops in traditional programming languages.

* GNU m4 without -G option will define the macro __gnu__ to expand to the empty string.

  On UNIX systems, GNU m4 without the -G option will define the macro __unix__, otherwise the macro unix. Both will expand to the empty string.

1.75 m4.guide/Concept index

```
                Concept index
*************
```

1.76 m4.guide/Macro index

Macro index
****

References are exclusively to the places where a builtin is introduced the first time. Names starting and ending with __ have these characters removed in the index.

```
changeword
  Changeword

debugfile
  Debug Output

debugmode
  Debug Levels

decr
  Incr

define
  Define

defn
  Defn

divert
  Divert

divnum
  Divnum

dnl
  Dnl

dumpdef
  Dumpdef

errprint
  Errprint

esyscmd
  Esyscmd

eval
  Eval

file
  Errprint

format
  Format

gnu
  Other Incompat

ifdef
  Ifdef

ifelse
  Ifelse

include
  Include
```

```
incr
  Incr

index
  Index

indir
  Indir

len
  Len

line
  Errprint

m4exit
  M4exit

m4wrap
  M4wrap

maketemp
  Maketemp

patsubst
  Patsubst

popdef
  Pushdef

pushdef
  Pushdef

regexp
  Regexp

shift
  Loops

sinclude
  Include

substr
  Substr

syscmd
  Syscmd

sysval
  Sysval

traceoff
  Trace

traceon
  Trace
```

```
translit
  Translit

undefine
  Undefine

undivert
  Undivert

unix
  Other Incompat
```