M4 Macros for Electric Circuit Diagrams in LaTeX Documents

Dwight Aplevich

Version 5.2

Contents

1 Introduction

> Before every conference, I find Ph.D.s in on weekends running back and forth from their offices to the printer. It appears that people who are unable to execute pretty pictures with pen and paper find it gratifying to try with a computer [9].

This document describes a set of macros, written in the **m4** macro language [7], for producing electric circuits and other diagrams in LaTeX documents. The macros evaluate to drawing commands in **pic**, a line-drawing language [8] which is readily available and quite simple to learn. The result is a system with the advantages and disadvantages of TeX itself, since it is macro-based and non-wysiwyg, and since it uses ordinary character input. The book from which the above quotation is taken correctly points out that the payoff is in quality of diagrams, at the price of the time spent in learning how to draw them.

A collection of basic components and conventions for their internal structure are described. For particular drawings it is often convenient to create new macros, or combinations of them, using consistent conventions, so macros such as these are only a starting point. The IEEE standard [6] has been followed. The macros described here make extensive use of the characteristics of **pic** and have been designed, where possible, to be an extension of the language.

2 Using the macros

The diagram source file is preprocessed as illustrated in Fig. 1. The source, together with the predefined macros, is first passed through **m4**, and then through a **pic** interpreter that produces a `.tex` file to be inserted into the `.tex` source using the `\input` command.

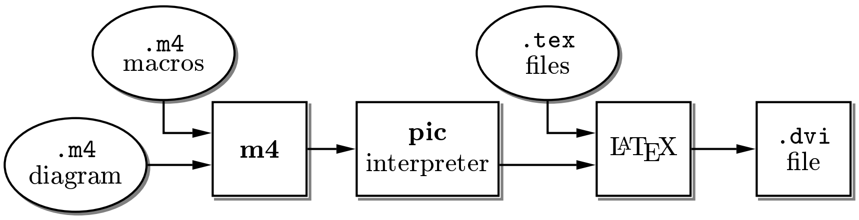


Figure 1: Inclusion of figures and macros in the LaTeX document

The `.tex` file contains, for example, **tpic** specials, LaTeX graphics, or other graphics commands, such as for **PSTricks** [14], which LaTeX will convert or include in a `.dvi` file. The `.dvi` file is then to be viewed or printed by a driver capable of interpreting any `\special` commands inserted by the **pic** processor.

To convert **pic** to TeX input one can use [3] **gpic -t** with a printer driver that understands **tpic** specials, typically [11] **dvips**. In some installations, **gpic** is simply named **pic,** but make sure that GNU **pic** [3] is being invoked rather than the older Unix **pic.** An alternative is **dpic**, as described later in this document. **Pic** processors contain basic macro facilities, so some of the concepts applied here require only a **pic** processor.

With judicious use of macros the features of both **m4** and **pic** can be exploited. The fastidious reader might observe that there are 3 languages being scrambled: **m4**, **pic**, and the **tpic, tex** or other output, not to mention the meta-language of the macros, and that this mixture might be a problem, but experience implies otherwise.

2.1 Quick start

The contents of file `quick.m4` are shown below to illustrate the language, to show several ways for placing circuit elements, and to give you enough information to produce basic labelled circuits such as Figure 2.

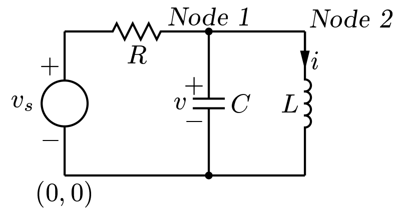


Figure 2: Diagram produced by file `quick.m4`

```
.PS                             # Pic input begins with .PS
cct_init                        # Set defaults

elen = 0.75                     # Variables are allowed
Origin: Here                    # Position names are capitalized
   source(up_ elen); llabel(-,v_s,+)
   resistor(right_ elen);  rlabel(,R,)

Node1: dot                      # Object names are capitalized
Node2: Node1+(elen*2/3,0)       # Named position relative to Node1

   {                            # Save current position and direction
      capacitor(down_ to (Here,Origin))     #(Here,Origin) = (Here.x,Origin.y)
      rlabel(+,v,-); llabel(,C,)
      dot
      }                         # Restore position and direction

   line to Node2
   inductor(down_ Here.y-Origin.y); rlabel(,L,); b_current(i)
   line to Origin

# Omit these if the labels are not wanted:
   "$(0,0)$" at Origin below
   "\sl Node 1" at Node1 above
   "\sl Node 2" at Node2 above ljust
.PE                             # Pic input ends
```

To process the file, make sure that the libraries **libcct.m4** and **libgen.m4** are accessible. Verify that **m4** is installed. Now there are at least two possibilities, as follows, with slightly simpler usage to be given in Section 2.1.3.

2.1.1 Processing with gpic

If you are using **gpic** (on some systems the **gpic** command is `pic`), do the following. Type

```
m4 <path>libcct.m4 quick.m4 > quick.pic
gpic -t quick.pic > quick.tex
```

where `<path>` is the path to the `libcct.m4` file.

At the place in the LaTeX source where the figure is to be included, put the lines

```
\begin{figure}[hbt]
   \input quick
   \centerline{\box\graph}
   \caption{Customized caption for the figure}
   \label{Symbolic_label}
   \end{figure}
```

where the caption and label are customized for the figure. Then LaTeX the document.

2.1.2 Processing with dpic

If you are using **dpic** with the PSTricks macros, the commands are

```
m4 <path>pstricks.m4 <path>lib/libcct.m4 quick.m4 > quick.pic
dpic -p quick.pic > quick.tex
```

and the document should have the statement `\usepackage{pstricks}` in the header. The figure inclusion statements are

```
\begin{figure}[hbt]
   \centering
   \input quick
   \caption{Customized caption for the figure}
   \label{Symbolic_label}
\end{figure}
```

In both cases, the essential line is `\input quick` which inserts the previously-created file `quick.tex`.

2.1.3 Simplifications

If appropriate `include()` statements are placed at the top of the file `quick.m4`, then the **m4** commands illustrated above can be shortened to

```
m4 quick.m4 > quick.pic
```

For example, the following two lines can be inserted before the line containing `.PS`:

```
include(<path>pstricks.m4)
include(<path>libcct.m4)
```

where `<path>` is the path to the folder containing the libraries. Only the second line is necessary if **gpic** is used or if the libraries were installed so that PSTricks is assumed by default. On some systems, setting the environment variable `M4PATH` to the library folder allows the above lines to be simplified to

```
include(pstricks.m4)
include(libcct.m4)
```

In the absence of a need to examine the file `quick.pic`, the commands for producing the `.tex` file can be reduced to

```
m4 quick.m4 | dpic -p > quick.tex
```

When many files are to be processed, then a facility such as Unix **make**, which is also available in several PC versions, can be used to automate the manual commands given above. On systems without such a facility, a scripting language can be used, or you can put several diagrams into a single source file so that they get processed together. Put each diagram in the body of a LaTeX macro, as shown:

```
\newcommand{\diaA}{%
.PS
drawing commands
.PE
\box\graph }%  \box\graph not required for dpic
\newcommand{\diaB}{%
.PS
drawing commands
.PE
\box\graph }%  \box\graph not required for dpic
```

Process the file using **m4** and **dpic** or **gpic** to produce a `.tex` file, insert this into the LaTeX source using `\input`, and invoke the macros at the appropriate places. If you include more than one diagram in a source file, remember that macro definitions are preserved between diagrams.

3 Pic essentials

Pic source is a sequence of lines in a file. Each diagram usually corresponds to a separate file, but this is not essential. The first line of a diagram begins with `.PS` with optional following arguments, and the last line is normally `.PE`. Lines outside of these are passed through the **pic** processor unchanged.

The visible objects can be divided conveniently into two classes, the *linear* objects `line, arrow, spline, arc,` and the *planar* objects `box, circle, ellipse.`

The object `move` is linear but draws nothing. A composite object, or `block,` is planar and is a set of simpler objects contained within square brackets: `[` *objects* `]`.

Text strings, typically meant to be typeset by LaTeX, have the double quote character at the beginning and end. Their typeset dimensions are unknown in advance, but can be obtained as demonstrated in Section 10.

Pic allows the definition of real-valued variables, which are alphameric names beginning with lower-case letters, and computations using them.

Finally, positions on the diagrams can be defined, either indpendently or relative to drawn objects. The name of a position begins with an upper-case letter.

3.1 Manuals

At the time of writing, the classic **pic** manual [8] can be obtained from URL:

`ftp://cm.bell-labs.com/cm/cs/cstr/116.ps.gz`

A more complete manual [10] is included in the GNU **groff** package. Compressed postscript versions of both are available from

`http://ece.uwaterloo.ca/~aplevich/dpic/`

In both of the above, explicit use of ***roff** string and font constructs should be replaced by their LaTeX equivalents as necessary. Further explanation is available, for example, from the **gpic** 'man' page, part of the GNU **groff** package.

Examples of use of the circuit macros in an electronics course are available on the web [2].

For a discussion of the use of "little languages" in document production, and of **pic** in particular, see Chapter 9 of [1]. Chapter 1 of [4] also contains a brief discussion of this and other languages. Section 10 of this document describes how to overcome the problem mentioned in [4] of determining the dimensions of typeset text in diagrams.

3.2 The linear objects: `line, arrow, spline, arc`

A line can be drawn as follows:

`line from` *position* `to` *position*

where *position* is defined below, or

`line` *direction distance*

where *direction* is one of `up, down, left, right.` When used with the **m4** macros described here, it is preferable to add an underscore: `up_, down_, left_, right_.` The *distance* is a number or expression, and the units are inches, but the assignment

`scale = 25.4`

has the effect of changing the units to millimetres, as described in Section 8.

Lines can also be drawn to any distance in any direction. The example,

`line up_ 3/sqrt(2) right_ 3/sqrt(2)`

draws a line 3 units long from the current location, at a 45° angle above horizontal.

The above methods of specifying the direction and length of a line are referred to as a *linespec* in this document.

Lines can be concatenated. For example to draw a triangle:

`line up_ sqrt(3) right_ 1 then down_ sqrt(3) right_ 1 then left_ 2`

A *position* can be defined by a coordinate pair, e.g. `3,2.5`, more generally using parentheses by `(`*expression, expression*`)`, or by the construction `(`*position, position*`)`, the latter taking the x-coordinate from the first position and the y-coordinate from the second. A position can be given a symbolic name beginning with an upper-case letter, e.g. `Top: (0.5,4.5)`. Such a definition does not affect the calculated figure boundaries. The current position `Here` is always defined. The coordinates of a position are accessible, e.g. `Top.x` and `Top.y` can be used in expressions. The center, start, and end of linear objects are valid positions, for example:

```
line from last line.start to 2nd last arrow.end then to 3rd line.center
```

which also illustrates how to refer to a previously-drawn element if it has not been given a name.

Objects can be named (using a name commencing with an upper-case letter), for example:

```
Bus23:  line up right
```

after which positions associated with the object can be referenced using the name, for example:

```
arc cw from Bus23.start to Bus23.end with .center at Bus23.center
```

An arc is drawn by specifying its rotation, starting point, end point, and center, but sensible defaults are assumed if any of these are omitted.

The linear objects can be given arrowheads at the start, end, or both ends, for example:

```
line dashed <- right 0.5
arc <-> height 0.06 width 0.03 ccw from Here to Here+(0.5,0) \
   with .center at Here+(0.25,0)
spline -> right 0.5 then down 0.2 left 0.3 then right 0.4
```

The arrowheads on the arc above have had their shape adjusted using the `height` and `width` parameters.

Finally, lines can be specified as `dotted`, `dashed`, or `invisible`, as in the above example.

3.3 The planar objects: `box`, `circle`, `ellipse`

The planar objects are drawn by specifying the width, height, and position of the center, thus:

```
A: box ht 0.6 wid 0.8 at (1,1)
```

after which, in this example, the position `A.center` is a defined position, and can be written simply as `A`. In addition, the compass corners `A.n`, `A.s`, `A.e`, `A.w`, `A.ne`, `A.se`, `A.sw`, `A.nw`, are all defined, as are the dimensions `A.height` and `A.width`. For example, two touching circles can be drawn as shown:

```
circle radius 0.2
circle diameter (last circle.width * 1.2) with .sw at last circle.ne
```

The planar objects can be filled with gray by the `fill` *number* parameter, where *number*= 0 means black, and *number*= 1 means white. Omitting the number produces a medium gray. Thus, for example,

```
box dashed fill
```

produces a gray dashed box.

Colours and more elaborate line and fill styles are not part of the basic **pic** language, but can be incorporated, depending on the printing device, by inserting `\special` commands or other lines beginning with a backslash in the drawing code. In fact, arbitrary lines can be inserted into the output using

`command "`*string*`"`

where *string* is the line to be output.

3.4 Compound objects

A group of statements enclosed in square brackets can be placed as if it were a box. Thus, the code fragment shown is found in a large digital diagram:

```
Ands:  [ right_
   And1:  AND_gate
      line right_ del/2 then down_ del*3/2 \
         then left_ And1.Out.x-And1.In1.x+del then down_ del then right_ del/2
   And2:  AND_gate with .In1 at Here
      line from And2.Out right_ del/2 then down_ del then right_ del/2
   ...  ]  with .And2.In1 at (K.x+2*del,IC5.Pin9.y)
```

In the above, each of the gate macros evaluates to a composite object in which the positions `Out`, `In1`, and others are defined. Several gates and connecting lines are contained in a block that is placed such that position `In1` within `And2` is at a predefined position.

3.5 Other language elements

All objects have default sizes, directions, and other characteristics, so part of the specification of an object can sometimes be profitably omitted.

Another possibility for defining positions is

expression `of the way between` *position* `and` *position*

which is abbreviated as

expression `<` *position* `,` *position* `>`

but care has to be used in processing the latter construction with **m4**, since the comma may have to be put within quotes, `` `,' `` to distinguish it from the `m4` argument separator.

Positions can be calculated using expressions containing variables. The scope of a position is the current block. Thus, for example,

```
theta = atan2(B.y-A.y,B.x-A.x)
line to Here+(3*cos(theta),3*sin(theta)).
```

Expressions are the usual algebraic combinations of primary quantities: constants, environmental parameters such as `scale`, named variables, horizontal or vertical coordinates, using the constructs *position*`.x` or *position*`.y`, dimensions of **pic** objects, e.g. `last circle.rad`.

The logical operators `==, !=, <=, >=, >, <` apply to expressions, and strings can be tested for equality or inequality. A modest selection of numerical functions is also provided: the single-argument functions `sin, cos, log, exp, sqrt, int`, where `log` and `exp` are base-10, the binary functions `atan2, max, min`, and the random-number generator `rand()`.

A **pic** manual should be consulted for details and more examples, and for other facilities, such as the branching facility,

`if` *expression* `then {` *anything* `} else {` *anything* `}`,

the looping facility,

`for` *variable* = *expression* `to` *expression* `by` *expression* `do {` *anything* `}`,

operating-system commands, string handling, **pic** macros, and file inclusion. Local and global variables and elementary file facilities are available.

4 Two-terminal elements

All fundamental two-terminal element macros have been constructed according to the conventions described below.

First, all drawing macros have default arguments, so that only arguments different from default need be specified. The arguments are given in Section 14.

Consider the resistor shown in Fig. 3, which also serves as an example of **pic** commands.

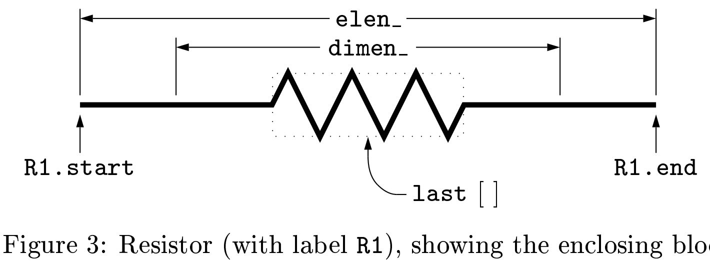


Figure 3: Resistor (with label `R1`), showing the enclosing block

The first part of the source for Fig. 3 is as follows:

```
.PS
   cct_init
   linewid = 2.0
   linethick_(2.0)
```

```
R1: resistor
```

4.1 Circuit and element basics

The lines of Fig. 3 and the remaining source lines of the file are explained below:

- The first line invokes an almost-empty macro that initializes local variables used by some circuit-element macros. This macro can be customized to set line thicknesses, maximum page sizes, scale parameters, or other global quantities as desired.

- By default, the size of two-terminal elements is a multiple of the **pic** variable `linewid`, which has initial value 0.5 in., so the value 2.0 is assigned to this variable to make a suitably big diagram for this manual. The body size of an element is determined by the macro `dimen_`, which evaluates by default to `linewid`, and the default element length is `elen_`, which is `dimen_*3/2` by default. These dimensions can all be customized. For resistors, the length of the body is `dimen_/2`, and the width is `dimen_/6`.

- The macro `linethick_` sets the thickness of subsequent lines (to 2.0pt in the example).

- The two-terminal element macros expand to sequences of drawing commands that begin with '`line invis` *linespec*', where *linespec* is the first argument of the macro if it is non-blank, otherwise by default the line is drawn a distance `elen_` in the current direction, which is to the right by default. The invisible line is first drawn, then the element is drawn on top of the line. The element—rather the initially-drawn invisible line—can be given a name, `R1` in the example, so that positions `R1.start` and `R1.end` are defined as shown and `R1.center` is also defined.

- The element body is enclosed by a block, which later can be used to place labels around the element. The block corresponds to an invisible rectangle with horizontal top and bottom lines, regardless of the direction in which the element is drawn. In the diagram a dotted box has been drawn to show the block boundaries.

- The last sub-element, identical to the first in each two-terminal element, is an invisible line that can be referenced later to place labels or other elements. This may be over-kill. If you create your own macros you might choose simplicity over generality, and only include visible lines.

To produce Fig. 3, the following embellishments were included after the previously-shown source:

```
   thinlines_
   box dotted wid last [].wid ht last [].ht at last []

   spline <- down 0.2 then down 0.1 right 0.1 then right 0.1 \
      from last [].s
   "{\tt last} [ ]" ljust

   arrow <- down 0.2 from R1.start+(0,-0.05); "\tt R1.start" below
   arrow <- down 0.2 from R1.end+(0,-0.05); "\tt R1.end" below

   dimension_(from R1.start to R1.end,0.45,\tt elen\_,0.4)
   dimension_(right_ dimen_ from R1.c-(dimen_/2,0),0.3,
      \tt dimen\_,0.5)
.PE
```

- The line thickness is set to the default thin value of 0.4pt, and the box displaying the element body block is drawn. Notice how the width and height can be specified, and the box centre positioned at the centre of the block.

- The next paragraph draws two objects, a spline with an arrowhead, and a string left justified at the end of the spline. Other string-positioning modifiers than `ljust` are `rjust`, `above`, and `below`. Lines to be read by **pic** can be continued by putting a backslash as the rightmost character.

- The last paragraph invokes a macro for dimensioning diagrams. Lines can be broken before a macro argument because **m4** ignores white space before arguments.

4.2 The two-terminal elements

Figs. 4 and 5 are tables of the two-terminal elements. Several elements are included more than once to illustrate some of their arguments. The arguments for each element are listed in Section 14.

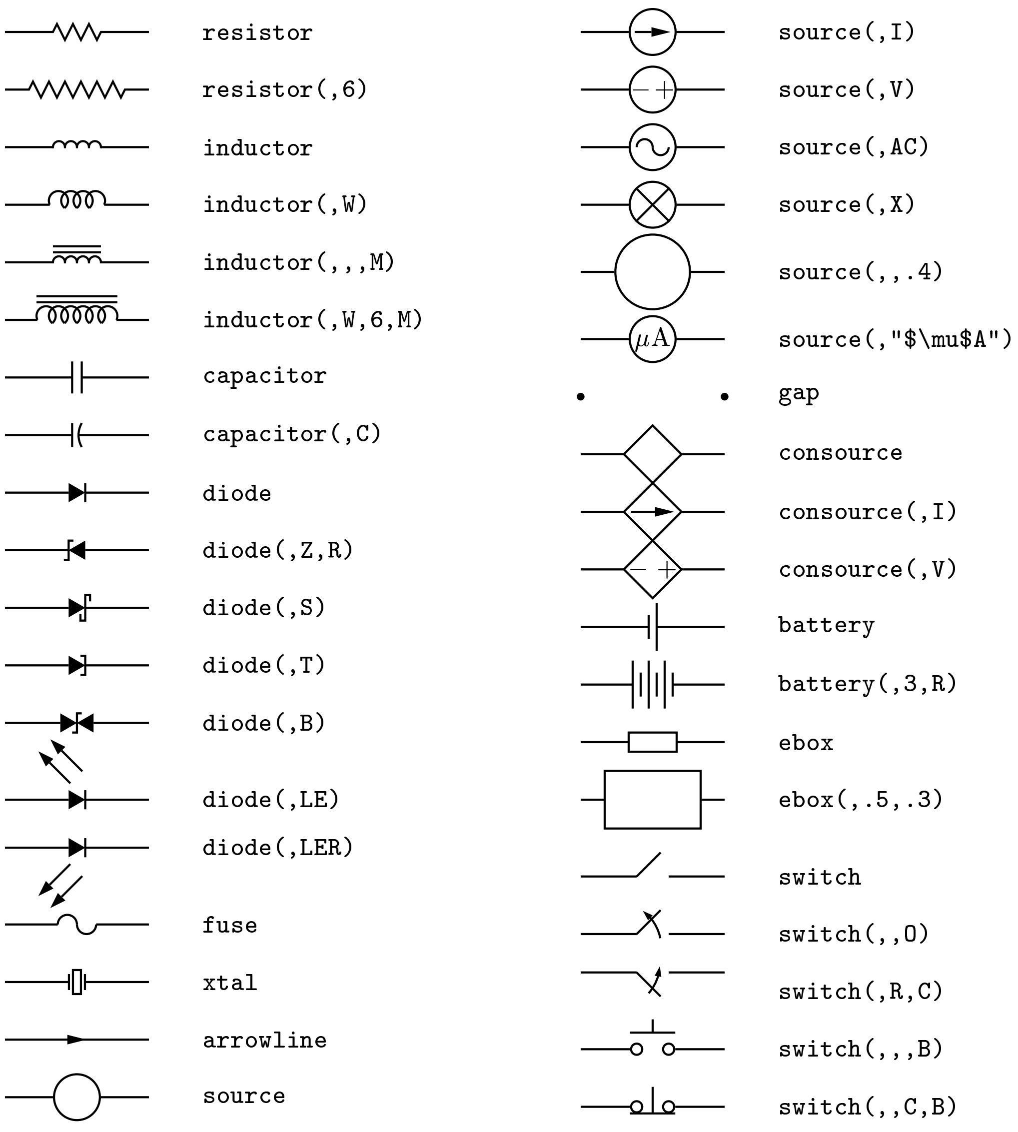


Figure 4: Two-terminal element macros, with some variations

In the **m4** language, macro arguments are included within parentheses following the macro name, with no space between the name and the opening parenthesis.

The first argument of the two-terminal elements, if included, defines the line direction and length along which the element is drawn. The other arguments produce variants of the default elements. Thus, for example,

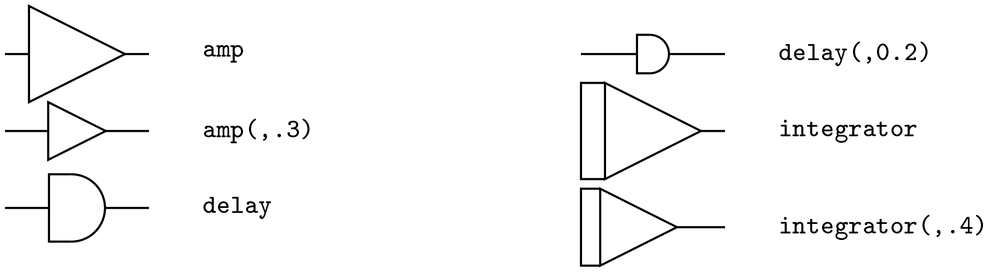


Figure 5: Amplifier, delay, and integrator

`resistor(up_ 1.25,7)`
draws a resistor 1.25 units long up from the current position, with 7 vertices per side. The name `up_` is a macro that resets the current macro and **pic** directional parameters to point up.

Most of the two-terminal elements are oriented; they have a defined polarity. Several, such as `battery` or `diode`, have an argument that reverses polarity, but there is also a more general mechanism. The first argument of the macro
`reversed(`'*macro name*'`,`*macro arguments*`)`
is the name of a two-terminal element in quotes, followed by the element arguments. The element is drawn with reversed direction. Thus,
`diode(right_ 0.4); reversed('diode',right_ 0.4)`
draws two diodes to the right, but the second one points left.

4.3 Branch-current arrows

The macro
`b_current(`*label*`, above_|below_, In|Out, Start|End)`
draws an arrow from the start of the last-drawn two-terminal element toward the body. If the fourth argument is `End`, the arrow is drawn from the end of the element toward the body. If the third element is `Out`, the arrow is drawn out, away from the body. The first argument is an optional

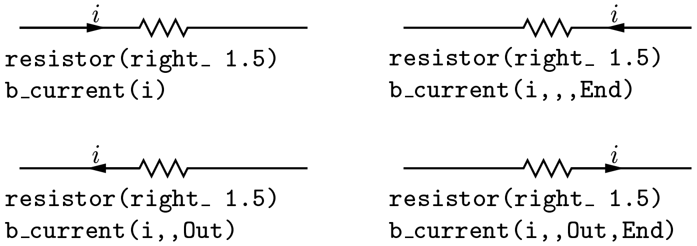


Figure 6: Illustrating `b_current`

label, of which the default position is the macro `above_`, which evaluates to `above` if the current direction is right, or to `ljust, below, rjust` if the current direction is respectively down, left, up. A non-blank second argument specifies the relative position, for example `below_`, which places the label below with respect to the current direction, or an absolute position, for example `below`, or `ljust`, with respect to the arrowhead.

4.4 Labels

Macros for labelling two-terminal elements are included:
`llabel(` *arg1,arg2,arg3* `)`
`clabel(` *arg1,arg2,arg3* `)`
`rlabel(` *arg1,arg2,arg3* `)`
`dlabel(` *long,lat,arg1,arg2,arg3* `)`

The first macro places the three arguments, which are treated as math-mode strings, on the left side of the element block *with respect to the current direction:* `up, down, left, right.` The second places the arguments along the centre, and the third along the right side. Thus a simple circuit example with labels is

```
% `Loop+.m4'
.PS
cct_init
define(`dimen_',0.75)
loopwid = 1; loopht = 0.75
  source(up_ loopht); llabel(-,v_s,+)
  resistor(right_ loopwid); llabel(,R,); b_current(i)
  inductor(down_ loopht,W); rlabel(,L,)
  capacitor(left_ loopwid,C); llabel(+,v_C,-); rlabel(,C,)
.PE
```

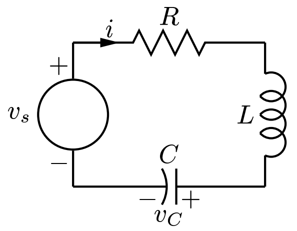


Figure 7: A loop with labelled elements

which produces Fig. 7. The macro `dlabel` performs the above functions for an obliquely-drawn element, placing the three macro arguments at `vec_(-long,lat)`, `vec_(0,lat)`, and `vec_(long,lat)` respectively relative to the centre of the element. Labels beginning with a double quote are copied literally rather than assumed to be in math mode.

5 Other circuit elements

Many basic elements are not two-terminal. Fig. 8 shows an operational amplifier, variants of the transformer macro, and some ground symbols. The ground symbol macro has three arguments:
`ground( at` *position*`, T, N|F|S|L|P )`

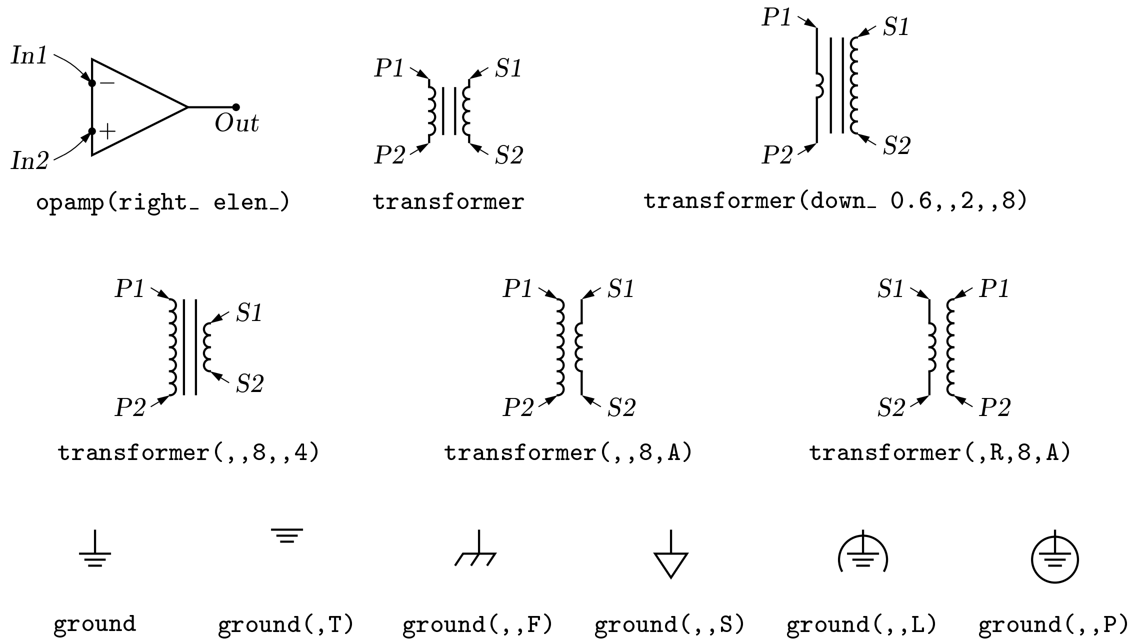


Figure 8: Miscellaneous basic elements

so that, for example, the lines

```
move to (1.5,2); ground
ground(at (1.5,2))
```

have identical effect. The second argument truncates the stem, and the third defines the symbol type.

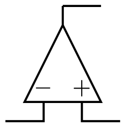
The opamp, transformer, and most other non-two-terminal elements are enclosed in a block, and therefore may be named. They generally contain named locations in the interior. An invisible line determining length and direction (but not position) can be specified by the first argument, as for the two-terminal elements. Instead of positioning by the first line, the enclosing block must be placed as shown: *element*`(at` *position*`)`, or by using its compass corners, thus: *element* `with` *corner* `at` *position*, or, when the block contains a predefined location, thus: *element* `with` *location* `at` *position*.

The operational amplifier, with macro

`opamp(` *linespec*, -, +)

is enclosed in a block containing the three predefined internal locations shown on the diagram, *In1*, *In2*, and *Out*, which can be referenced in later commands, for example as '`last [].Out`.' The first argument defines the direction and length of the opamp, but the position is determined either by predefined enclosing block of the opamp, or by a construction such as '`opamp with .In1 at Here`' which places the internal position *In1* at the specified location. There are optional second and third arguments for which the defaults are `$-$` and `$+$` respectively. For example, the following code fragment places an opamp with three connections as shown:

```
line right 0.2 then up 0.1
A: opamp(up_,,,.4) with .In1 at Here
  line right .2 from A.Out
  line down .1 from A.In2 then right .2
```



The transformer macro is

`transformer(` *linespec*, `L|R, np, A, ns )`

and has predefined internal locations *P1*, *P2*, *S1*, and *S2*. The first argument specifies the direction and distance from *P1* to *P2*, with position determined by the enclosing block as for opamps. The second argument places the secondary side of the transformer to the left or right of the drawing direction. The optional third argument specifies the number of primary arcs, the fourth omits the iron core, and the fifth specifies the number of secondary arcs. A transformer with four connections is illustrated as follows:

```
line right 0.2
up_
A: transformer(,R) with .P1 at Here
  line left .2 from A.P2
  line right .2 from A.S2 then up .1
  line right .2 from A.S1 then down .1
```

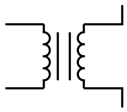


Fig. 9 illustrates the seven-argument `nport` macro. The first argument is a box specification, such as size or fill parameters, or text. The second to fifth arguments specify the number of ports (pin-pairs) to be drawn respectively on the west, north, east, and south sides of the box. The end of each pin has a name corresponding to the side, port number and a or b pin, as shown. The sixth argument specifies the ratio of port width to inter-port space, and the last argument is the pin length. The complete structure is enclosed in a block.

Fig. 10 shows some variants of the macro

`contact(O|C, R)`

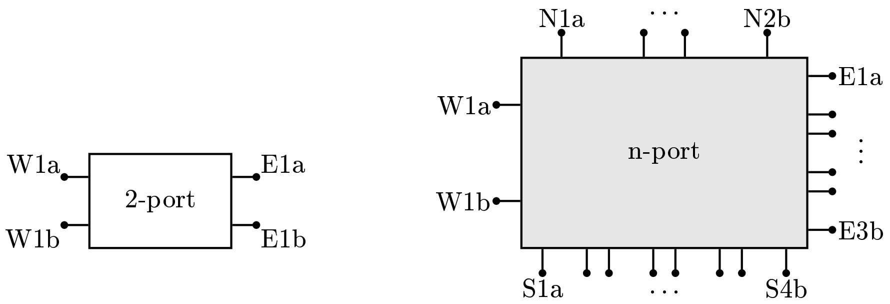
which contains predefined locations *P*, *C*, *O* for the armature and normally closed and normally open terminals. The macro

`relay(`*poles*`, O|C, R)`

defines coil terminals *V1*, *V2* and contact terminals P_i, C_i, O_i.

Fig. 11 shows the variants of bipolar transistor macro

`bi_tr(` *linespec*, `L|R, P, E)`

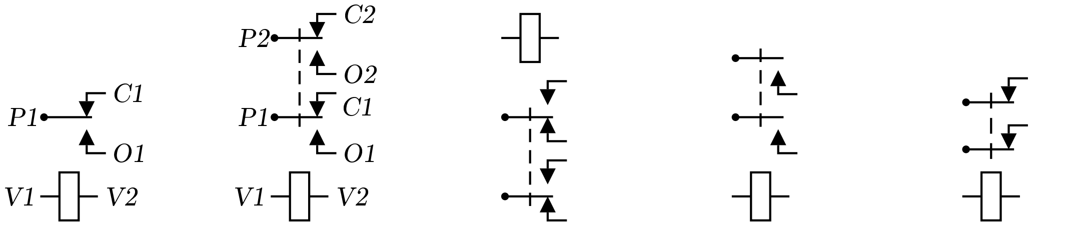


`nport("2-port")`

`nport(wid 1.5 ht 1 fill_(.9) "n-port",1,2,3,4)`

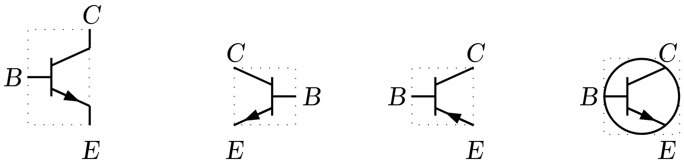
Figure 9: The `nport` macro draws a sequence of pairs of named pins on each side of a box. The default is a twoport. The pin names are shown.

`contact` `contact(,R)` `contact(O,)` `contact(C,)`



`relay` `relay(2,,)` `relay(2,,R)` `relay(2,O,)` `relay(2,C,)`

Figure 10: Variants of the `contact` and `relay` macros (default direction right)

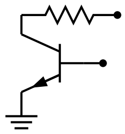


`bi_tr(up_ dimen_)` `bi_tr(,R)` `bi_tr(,,P)` `bi_tr(,,,E)`

Figure 11: Bipolar transistor variants (current direction upward)

which contains predefined internal locations *E*, *B*, *C*. The first argument defines the distance and direction from *E* to *C*, with location determined by the enclosing block as for other elements, and the base placed to the left or right of the current drawing direction according to the second argument. Setting the third argument to '`P`' creates a PNP device instead of NPN, and setting the fourth to '`E`' draws an envelope around the device. Thus for example, the following code fragment places a bipolar transistor and connects a ground to the emitter and a resistor to the collector as shown:

```
dot; line left_ .1; up_
Q1: bi_tr(,R) with .B at Here
ground(at Q1.E)
line up .1 from Q1.C; resistor(right_ dimen_); dot
```



Some FETs with predefined internal locations *S*, *D*, and *G* are also included, with similar arguments to those of `bi_tr`, as shown in Fig. 12. The macros in the figure are wrappers for the general `smosfet` and `mosfet` macros. The latter has an option for a disconnected substrate, illustrated in Fig. 13. Similarly a UJT macro with predefined internal locations *B1*, *B2*, and *E* is illustrated in Fig. 14, and an SCR macro with predefined internal locations *T1*, *T2*, and *G*

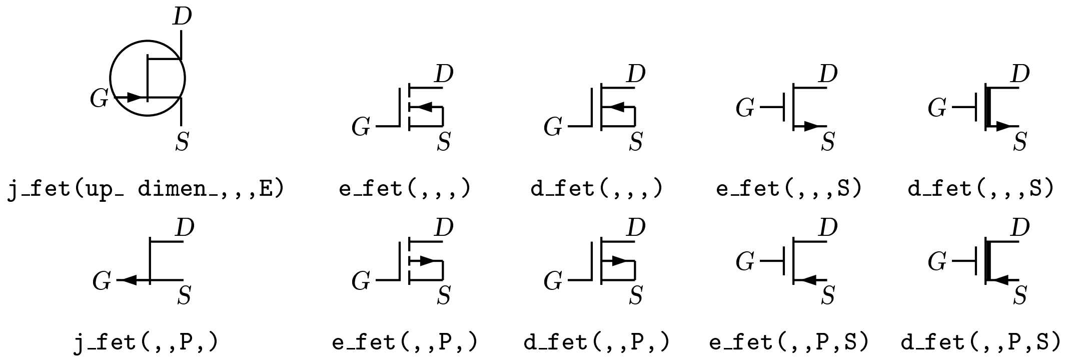


Figure 12: JFET, insulated-gate enhancement and depletion MOSFETS, and simplified versions of the MOSFETS, see [12]. These macros are wrappers that invoke the `mosfet` and `smosfet` macros.

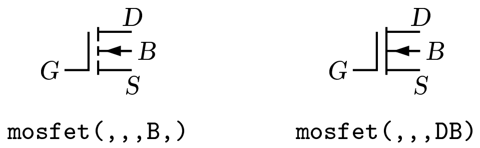


Figure 13: The macro `mosfet(`*linespec*`,L|R,P,[D][B],E)` has a disconnected-substrate option.

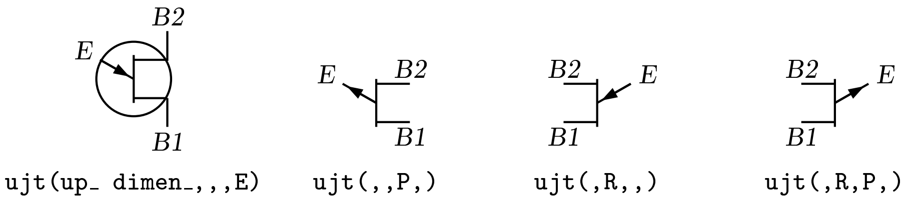


Figure 14: UJT variants, with current drawing direction up.

is illustrated in Fig. 15. The number of possible semiconductor symbols is very large, so these macros must be regarded as prototypes. Some other non-two-terminal macros are `dot` which has an optional argument '`at` *location*', the line-thickness macros, the `fill_` macro, and `crossover`, which is a useful if archaic method to show non-touching conductor crossovers, as in Figure 16.

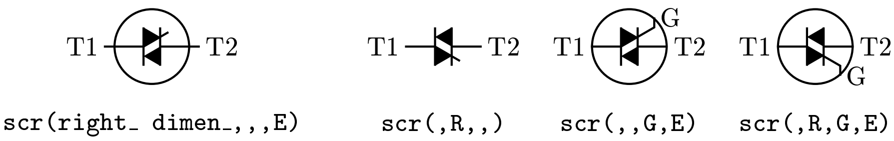


Figure 15: SCR variants, drawing direction to the right.

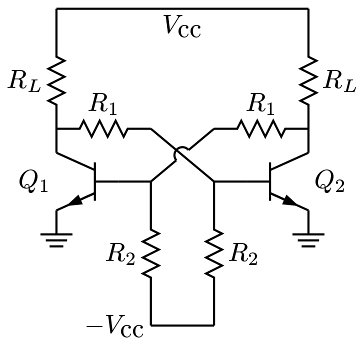


Figure 16: Bipolar transistor circuit, illustrating `crossover`

6 Directions

Aside from its block-structure capabilities, looping, and macros, **pic** has a very useful concept of the current point and current direction, the latter unfortunately limited to `up, down, left, right.` Objects can be drawn at absolute locations or placed relative to previously-drawn objects. These macros need to know the current direction so whenever `up, down, left, right` are used they should be written as `up_, down_, left_, right_` which are macros.

To draw circuit objects in other than the standard four directions, the macros `Point_(`*degrees*`)`, `point_(`*radians*`)`, and `rpoint_(`*rel linespec*`)` re-define the direction-cosine macros `x_, y_` which are used within element-drawing macros. Thus '`Point_(-30); resistor`' draws a resistor along a line with slope -30 degrees. '`rpoint_(to Z)`' sets the current direction cosines to point to location Z. Macro `vec_(x,y)` evaluates to the position `(x,y)` rotated by the argument of the previous `Point_`, `point_` or `rpoint_` command. The macro `rvec_(x,y)` evaluates to position `Here + vec_(x,y)` and is the principal device used to define relative locations in the circuit macros. Thus, `line to rvec_(x,0)` draws a line of length `x` in the current direction.

Fig. 17 shows a circuit drawn using the above macros. The source for the figure is below, and illustrates that some hand-placement of labels using `dlabel` may be useful when elements are drawn other than horizontal or vertically.

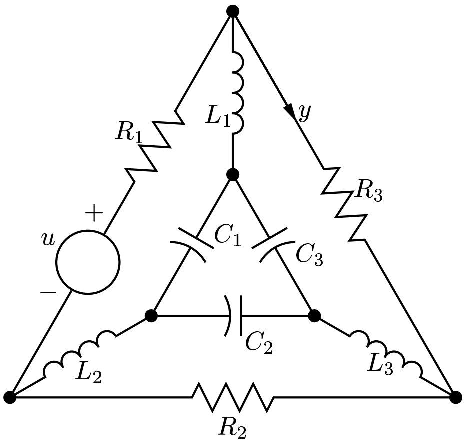


Figure 17: Illustrating elements drawn at oblique angles

```
% ex02.m4
.PS
define(‘elen_’,‘dimen_’)          # short elements
define(‘sourcerad_’,‘dimen_*0.2’)
linewid = 0.85

Ct:dot
   Point_(-60); capacitor(,C); dlabel(0.14,0.14,,,C_3)
Cr:dot
   left_; capacitor(,C); dlabel(0.14,0.14,C_2,,)
Cl:dot
   down_; capacitor(from Ct to Cl,C); dlabel(0.14,0.14,C_1,,)

T:dot(at Ct+(0,elen_))
   inductor(from T to Ct); dlabel(0.12,-0.1,,,L_1)

   Point_(-30); inductor(from Cr to Cr+vec_(elen_,0))
      dlabel(0,-0.07,,L_3,)
R:dot
L:dot( at (Cl-(Cos(30)*(elen_),0),R) )

   inductor(from L to Cl); dlabel(0,-0.12,,L_2,)
```

```
   right_; resistor(from L to R); rlabel(,R_2,)
   move down 0.3

   resistor(from T to R); dlabel(0,0.15,,R_3,) ; b_current(y,ljust)

   line from L to 0.2<L,T>
   source(to 0.5 between L and T); dlabel(sourcerad_+0.07,0.1,-,,+)
      dlabel(0,sourcerad_+0.07,,u,)
   resistor(to 0.8 between L and T); dlabel(0,0.15,,R_1,)
   line to T
.PE
```

Since **m4** macro arguments are separated by commas, any commas that are integral parts of the arguments must be protected either by parentheses as illustrated in `inductor(from Cr to Cr+vec_(elen_,0))`, or by multiple single quotes, ``` ``,'' ```, as necessary. Commas also may be avoided by writing `0.5 between L and T` instead of `0.5<L,T>`.

7 Logic gates

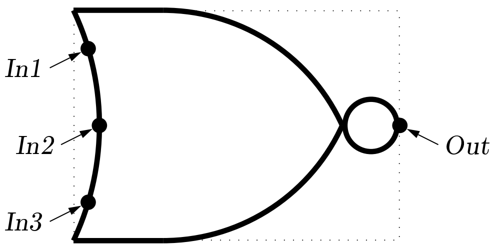


Figure 18: Illustrating `NOR_gate(3)`

Fig. 18 shows a 3-input NOR gate, with the enclosing block and predefined internal locations `Out, In1`–`In3`. Gate macros have an optional argument, an integer from 0 to N where N can be up to approximately 5, defining locations `In1,` $\cdots$ `In`N. By default $N = 2$, except for macros `NOT_gate` and `BUFFER_gate` which have one input `In1`, unless they are given an argument, which is treated as the line specification of a two-terminal element. Fig. 19 shows these and the other

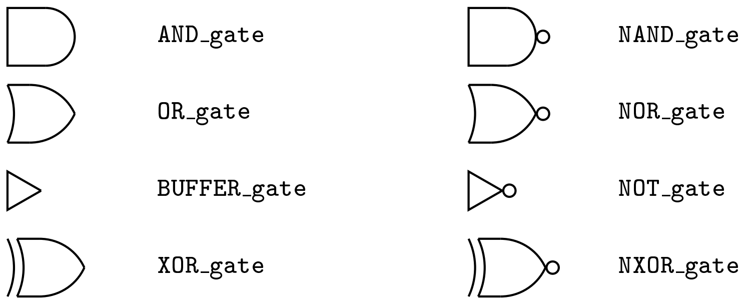


Figure 19: Basic logic gate library

basic logic gates included in library `liblog.m4`. These gates are typically not two-terminal elements and are normally drawn horizontally or vertically (although arbitrary directions may be set with e.g. `Point_(`*degrees*`)`). Each gate is contained in a block of typical height `6*L_unit` where `L_unit` is a macro intended to establish line separation for an imaginary grid on which the elements are superimposed.

The following source produces the SR flip-flop shown in Fig. 20 to illustrate change of direction. Note that when a gate is rotated, its input locations retain their positions relative to the gate body.

```
% ‘FF+.m4’
.PS
log_init
S: NOR_gate
  left_
R: NOR_gate at S+(0,-L_unit*9)
  line right_ L_unit*3 from S.Out ; line to (Here,R.In2) then to R.In2
  line left_ L_unit*3 from R.Out ; line to (Here,S.In2) then to S.In2
  line left_ 4*L_unit from S.In1 ; "$S$sp_" rjust
  line right_ 4*L_unit from R.In1 ; "sp_$R$" ljust
.PE
```

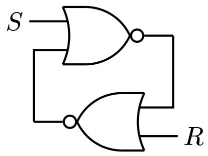


Figure 20: SR flip-flop

A good strategy for drawing complex logic circuits might be summarized as follows:

- Establish the absolute locations of gates and other major components (e.g. chips) relative to a grid of mesh size commensurate with `L_unit`, which is an absolute length.
- Draw minor components or blocks relative to the major ones, using parametrized relative distances.
- Draw connecting lines relative to the components and previously-drawn lines.
- Write macros for repeated objects.
- Tune the diagram by making absolute locations relative, and by tuning the parameters. Some useful macros for this are the following, which are in units of `L_unit`:

  `AND_ht, AND_wd`: the height and width of basic AND and OR gates

  `BUF_ht, BUF_wd`: the height and width of basic buffers

  `N_diam`: the diameter of NOT circles

A useful exercise might be to reproduce the example of Fig. 21.

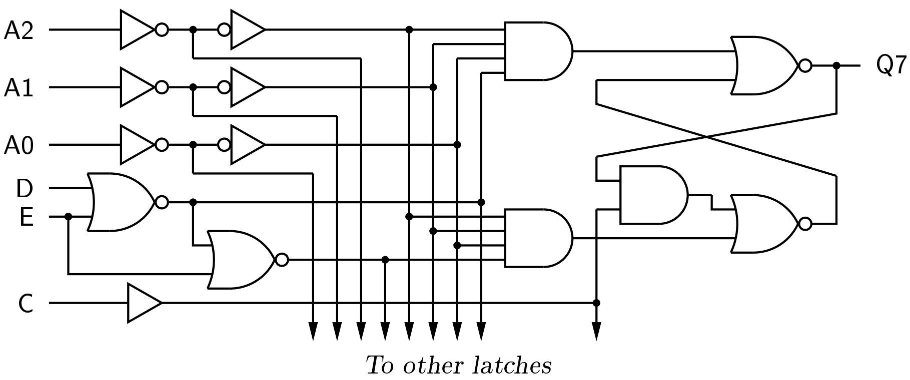


Figure 21: Multifunction latch circuit

In addition to the basic logic gates described here, some experimental flip-flop and IC chip diagrams are in the distributed example files.

8 Element and diagram scaling

There are several issues related to scale changes. You may wish to use millimetres, for example, instead of the default inches. You may wish to change the size of a complete diagram while keeping the relative proportions of objects within it. You may wish to change the sizes or proportions of individual elements within a diagram. You must take into account that line widths are scaled separately from drawn objects, and that the size of typeset text is independent of the **pic** language.

The **pic** scaling facilities will be described first, then the scaling of the defined circuit elements.

8.1 Pic scaling

There are at least three kinds of graphical elements to be considered:

1. The default sizes of linear and planar **pic** objects can be redefined by assigning values to the built-in **pic** variables `arcrad`, `arrowht`, `arrowwid`, `boxht`, `boxrad`, `boxwid`, `circlerad`, `dashwid`, `ellipseht`, `ellipsewid`, `lineht`, `linewid`, `moveht`, `movewid`, `textht`, `textwid`. The ···`ht` and ···`wid` parameters refer to the default sizes of vertical and horizontal lines, moves, etc., except for `arrowht` and `arrowwid` which refer to arrowhead dimensions. The `boxrad` parameter can be used to put rounded corners on boxes.

   Assigning a value to the variable `scale` multiplies all the built-in **pic** dimension variables except `arrowht`, `arrowwid`, `textht`, and `textwid` by the new value of `scale` (**gpic** multiplies them all). Thus the file `quick.m4` can be modified to use millimetres as follows:

   ```
   .PS                          # Pic input begins with .PS
   scale = 25.4                 # mm
   cct_init                     # Set defaults

   elen = 19                    # Variables can be used
   ...
   ```

   The `.PS` line can be used to scale the entire drawing, regardless of its interior. Thus, for example, the line `.PS 100/25.4` scales the entire drawing to a width of 100 mm.

   If the final picture width exceeds the value of `maxpswid`, which has a default size of 8.5, then the picture is scaled to this value. Similarly if the height exceeds `maxpsht`, (default 11), then the picture is scaled to fit.

2. The finished size of typeset text is independent of **pic** variables, but can be determined as in Section 10. Thus, once dimensions x and y are known, then `"text" wid` x `ht` y assigns the dimensions of `text`.

3. Line widths are independent of diagram and text scaling, and have to be set independently. For example, the assignment `linethick = 1.2` sets the default line width to 1.2pt. The macro `linethick_(`*points*`)` is also provided, together with default macros `thicklines_` and `thinlines_`.

8.2 Circuit scaling

The circuit elements all have default dimensions which are multiples of the **pic** environmental parameter `linewid`, so changing this parameter changes default element dimensions. The scope of a **pic** variable is the current block; therefore a sequence such as

```
resistor
[ linewid = linewid*1.5; resistor ]
resistor
```

produces a string of three resistors, the middle one larger than the other two. Alternatively, you may redefine the default length `elen_` or the body-size parameter `dimen_`. For example, adding the line

`define(‘dimen_’,dimen_*1.2)`

after the `cct_init` line of `quick.m4` produces slightly larger element body sizes.

9 Writing macros

Writing basic macros in the **m4** language is quite simple, and is described in numerous documents such as the original reference [7], or in later manuals [13]. If a new element is required, then modifying and renaming one of the library definitions or simply adding an option to it may suffice. Hints for drawing general two-terminal elements are given in `libcct.m4`. However if an element or composite is to be drawn in only one orientation then most of the elaborations used for general two-terminal elements in Section 4 can be dropped.

A macro is defined using quoted name and replacement text, as follows:

`define(‘`*name*`’,‘`*replacement text*`’)`

After this line is read by the **m4** processor, then whenever *name* is encountered as a separate string, it is replaced by its replacement text, which may have multiple lines. The quotation characters are used to defer macro expansion. Macro arguments are referenced inside a macro by number; thus `$1` refers to the first argument.

In the following example, two macros are defined to simplify the repeated drawing of a series resistor and series inductor, and the macro `tsection` defines a subcircuit that is replicated several times to generate Figure 22.

```
% ‘Tline.m4’
.PS
cct_init
hgt = elen_*1.5
ewd = dimen_*.9

define(‘sresistor’,‘resistor(right_ ewd); llabel(,r)’)
define(‘sinductor’,‘inductor(right_ ewd,W); llabel(,L)’)

define(‘tsection’,‘sinductor
  { dot; line down_ hgt*.25; dot
    gpar_( resistor(down_ hgt*.5); rlabel(,R),
          capacitor(down_ hgt*.5); rlabel(,C))
    dot; line down_ hgt*.25; dot }
  sresistor ’)

SW: Here
  gap(up_ hgt)
  sresistor
  for i=1 to 4 do { tsection }
  line dotted right_ dimen_/2
  tsection
  gap(down_ hgt)
  line to SW
.PE
```

10 Interaction with LaTeX

By using a bit of hackery, the dimensions of typeset text can be obtained and included in calculations within the diagram that contains the text. The difficulty is that font metrics are not known until LaTeX is invoked, and the solution is to process the diagram twice, just as symbolic references in source text require files to be processed twice. First the diagram `.m4` source is processed by **m4** and

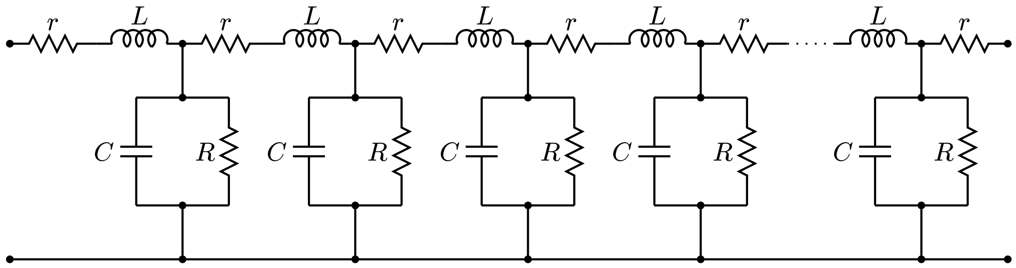


Figure 22: A lumped model of a transmission line, illustrating the use of custom macros.

a **pic** processor to make a `.tex` file, and the document source is LaTeXed to include the diagram and to write the required text dimensions into a supplementary file. Then the diagram `.m4` source and LaTeX document source are processed again.

The file `boxdims.sty` distributed with this package should be installed where LaTeX can find it, and should be invoked by `\usepackage{boxdims}` in the document source. The essential idea in `boxdims.sty` is to define a two-argument macro `\boxdims` which writes out definitions for the width, height and depth of its typeset second argument into file *jobname*`.dim`, where *jobname* is the name of the main source file. The first argument of `\boxdims` is used to construct unique symbolic names for these dimensions. Thus, the line

```
box "\boxdims{Q}{\Huge Hi there!}"
```

has the same effect as

```
box "\Huge Hi there!"
```

except that the line

```
define(‘Q_w’,77.6077pt__)define(‘Q_h’,17.27779pt__)define(‘Q_d’,0.0pt__)dnl
```

is written into file *jobname*`.dim` (and the numerical values depend on the current font).

The following small file, for example, produces the box shown in Figure 23:

```
% ‘eboxdims.m4’
.PS
sinclude(man.dim)  # The main input file is man.tex
box fill_(0.9) wid boxdim(Q,w) + 5pt__ ht boxdim(Q,v) + 5pt__ \
  "\boxdims{Q}{\large$\displaystyle\int_0^T e^{tA}dt$}"
.PE
```

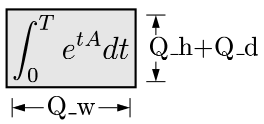


Figure 23: Fitting a box to typeset text.

First the source file for the figure is processed by **m4** and a **pic** interpreter to produce a `.tex` file, then LaTeX is run, and then these steps are repeated. In **m4,** `sinclude(`*jobname*`.dim)` will read the named file if it exists. The macro `boxdim(`*name,suffix,default*`)` from **libgen.m4** expands the expression `boxdim(Q,w)`, for example, to the value of `Q_w` if it is defined, else to its third argument if defined, else to 0, the latter two cases applying if *jobname*`.dim` doesn't exist yet. The values of `boxdim(Q,h)` and `boxdim(Q,d)` are similarly defined, and for convenience, `boxdim(Q,v)` evaluates to the sum of these. Macro `pt__` is defined as `*scale/72.27` in **libgen.m4**, to convert points to drawing coordinates.

In rare cases, the size of the diagram will affect page breaks, requiring the document source to be LaTeXed more than twice. To avoid this problem, the third argument of `boxdim` can be used to define an approximate initial size.

Another use of the `\boxdims` macro is in calculating the global dimensions of a diagram. Consider the following example:

```
.PS
B: box
  "Left text" at B.w rjust
  "Right text: $x^2$" at B.e ljust
.PE
```

The **pic** interpreter cannot know the dimensions of the text to the left and right of the box, and the resulting diagram is generated using the dimensions of the box alone. Modifying this example to

```
.PS
sinclude(jobname.dim)
B: box
  "\boxdims{L}{Left text}" wid boxdim(L,w) ht boxdim(L,v) at B.w rjust
  "\boxdims{R}{Right text: $x^2$}" wid boxdim(R,w) ht boxdim(R,v) at B.e ljust
.PE
```

assigns the correct width and height to the text strings, and the picture dimensions are then calculated correctly. The macros `s_init(`*name*`)` and `s_box(`*text*,*expr*,$\cdots$`)` are included to simplify the above example, as shown:

```
.PS
sinclude(jobname.dim)
s_init(unique name)
B: box
  s_box(`Left text') at B.w rjust
  s_box(`Right text: $x^%g$',2) at B.e ljust
.PE
```

The argument of `s_init` should be unique within *jobname*`.dim`. It is used to generate a unique `\boxdims` first argument for each invocation of `s_box` in the current file. If there are two or more arguments to `s_box` as in the second invocation of this macro above, then they are passed to the `sprintf()` function. If `s_init` has been omitted, the symbols "**!!**" are inserted into the text as a warning. Be sure to quote any commas in the arguments. Since the first argument is LaTeX, make a rule of quoting it to avoid comma and name-clash problems.

Further tricks can be played. The example

`S: s_box(`\includegraphics{`*file*`.eps}') with .sw at 0,0`

shows a nice way of including eps graphics in a diagram. When processed twice, the included picuture—defined by the string S in the example—has known position and dimensions, which can be used add vector graphics or text to the picture. To aid in overlaying planar objects, the macro `boxcoord(`*planar object*, *x-fraction*, *y-fraction*`)` evaluates to a position, with `boxcoord(`*planar object*`,0,0)` at the lower left corner of the object, and `boxcoord(`*planar object*`,1,1)` at its upper right.

11 PSTricks tricks

This section applies only to a **pic** processor (**dpic**) that is capable of producing PSTricks output. Arbitrary PSTricks commands can be mixed with **m4** input to create complicated effects, but some commonly required effects are particularly simple.

The rotation of text is illustrated by the file

```
% `Axes.m4'
.PS
  arrow right 0.7 "`$x$-axis'" below
  arrow up 0.7 from 1st arrow.start "`\rput[B]{90}(0,0){$y$-axis}'" rjust
.PE
```

which produces horizontal text, and text rotated 90° along the vertical line.

Another common requirement is the filling of arbitrary shapes, as illustrated by the following lines within a **.m4** file:

`command "‘\pscustom[fillstyle=solid,fillcolor=lightgray]{’"`
drawing commands for an arbitrary closed curve
`command "‘}%’"`

The macro `shade(`*gray value,closed line specs*`)` can be invoked to accomplish the same effect as the above example.

For colour printing or viewing, arbitrary colours can be chosen, as described in the **PSTricks** manual. **PSTricks** parameters can be set by inserting the line

`command "‘\psset{`*option = value,* ···`}’"`

in the drawing commands, or by using the macro `psset_(`*PSTricks options*`)`.

12 Developer's notes

Several years ago in the course of writing a book I took a few days off to write a **pic**-like interpreter **(dpic)** to automate the tedious coordinate calculations required by latex picture objects. The macros in this distribution and the interpreter are the result of that effort and of drawings I have had to produce since. The interpreter has been upgraded more recently to generate **mfpic**, **MetaPost** [5], **Postscript**, and **PSTricks** commands, the latter my preference because of its quality and flexibility, including facilities for colour and rotations, together with simple font selection. In addition, **xfig**-compatible output has been added. Instead of **pic** macros I preferred the equally simple but more powerful **m4** macro processor, and therefore **m4** is required here, although **dpic** now supports **pic**-like macros. Free versions of **m4** are available for Unix, Windows, and other operating systems.

If starting over today would I not just use one of the other drawing packages available these days? The answer is that **pic** is a good choice for the moderate geometrical calculations that are sometimes necessary in line drawings. The language is simple to learn, and more importantly to read, especially for backslash-phobics like me. The language has built-in looping and block-structure constructs that combine power with simplicity, and it has stood the test of time. However, no choice of tool is without compromise, and making good graphics is time-consuming no matter how it is done.

Using the **mfpic** output of **dpic** it is possible to produce Metafont alphabets of circuit elements or their subcomponents, thereby essentially removing dependence on device drivers, but with the complication of treating every graphic subcomponent as a TEX box. The **xfig** output allows elements to be defined and fine-tuned, and then using the interactive graphics of **xfig** to be quickly assembled into a circuit. Further refinement of the elements might still be required. **Dpic** will also issue low-level **MetaPost** or **Postscript** commands, so that diagrams defined using **pic** can be manipulated and combined with others. The user is responsible for ensuring that the correct fonts are provided.

13 Bugs

The distributed macros are not written for maximum robustness. Macro arguments could be tested for correctness and explanatory error messages could be written as necessary, but that would make the macros more difficult to read and to write. You will have to read them when unexpected results are obtained or when you wish to modify them.

Here are some hints, gleaned from experience and from comments I have received.

1. **Initialization:** If the first element macro evaluated is non-two-terminal or is within a **Pic** block, then later macros evaluated outside the block may produce the error message

   `there is no variable ‘rp_ang’`

   because `rp_ang` is not defined in the outermost scope of the diagram. To cure this problem, put the line

```
cct_init
```

immediately after the .PS line, or prior to the first block. It is entirely permissible to modify `cct_init` to include commonly-used diagram initializations, such as the `thicklines_` statement, and to invoke `cct_init` at the beginning of every diagram. For completeness, macros `gen_init, log_init, darrow_init` are also provided for cases where the circuit library is not needed.

2. **Pic objects versus macros:** A common error is to write something like

   ```
   line from A to B; resistor from B to C
   ```

   when it should be

   ```
   line from A to B; resistor(from B to C)
   ```

   This error is caused by an unfortunate inconsistency between the linear **pic** objects and the way **m4** passes macro arguments.

3. **Commas:** Remember that macro arguments are separated by commas, and commas that are part of an argument must be protected by parentheses or quotes. Thus,

   ```
   shadebox(box with .n at w,h)
   ```

   produces an error, whereas

   ```
   shadebox(box with .n at w`,'h)
   ```

   and

   ```
   shadebox(box with .n at (w,h))
   ```

   do not.

4. **Quotes:** Single quote characters are stripped in pairs by **m4**, so the string

   ```
   "``inverse''"
   ```

   will be typeset as if it were

   ```
   "`inverse'".
   ```

   The cure is to add single quotes.

   The most subtle part of writing **m4** macros is deciding when to quote arguments. In the context of circuits it seemed best to assume that macro arguments would not be protected by quotes at the level of macro invocation, but should be quoted inside each macro. There may be cases where this rule is not optimal.

5. **Dollar signs:** The i-th argument of an **m4** macro is `$`i, where i is an integer, so the following construction can cause an error when it is part of a macro,

   ```
   "$0$" rjust below
   ```

   since `$0` expands to the name of the macro itself. To avoid this problem, put the string in quotes, or write `"$`'0$"`.

6. **Name conflicts:** Using the name of a macro as part of a comment or string is a simple and common error. Thus,

   ```
   arrow right "$\dot x$" above
   ```

   produces an error message because `dot` is a macro name. Macro expansion can be avoided by adding quotes, as follows:

   ```
   arrow right `"$\dot x$"' above
   ```

   Library macros intended only for internal use have names that begin with `m4` to avoid name clashes, but in addition, a good rule is to quote all LaTeX in the diagram input.

   If extensive use of strings that conflict with macro names is required, then one possibility is to replace the strings by macros to be expanded by LaTeX , for example the diagram

```
.PS
 box "\stringA"
.PE
```

with the LaTeX macro

```
\newcommand{\stringA}{
Circuit containing planar inductor and capacitor}
```

7. **Current direction:** Some macros, particularly those for labels, do unexpected things if care is not taken to preset the current direction using macros `right_`, `left_`, `up_`, `down_`, or `rpoint_(·)`. Thus for two-terminal macros it is good practice to write, e.g.

   `resistor(up_ from A to B); rlabel(,R_1)`

   rather than

   `resistor(from A to B); rlabel(,R_1)`,

   which produce different results if the last-defined drawing direction is not `up`. It might be possible to change the label macros to avoid this problem without sacrificing ease of use.

8. **Pic error messages:** Some errors are detected only after scanning beyond the end of the line containing the error. The semicolon is a logical line end, so putting a semicolon at the end of lines may assist in locating bugs.

9. **Incompatible processors:** If you switch between e.g. **dpic** and **gpic,** remember that the libraries are set up to use **gpic** by default, otherwise `pstricks.m4` or `mfpic.m4` have to be processed before the other libraries. To redefine the default behaviour, change the `include` statements near the top of the libraries.

10. **Scaling: Pic** and these macros provide several ways to scale diagrams and elements within them, but subtle unanticipated effects may appear. The line `.PS` x provides a convenient way to force the finished diagram to width x. However if **gpic** is the **pic** processor then all scaled parameters are affected, including those for arrowheads and text, which may not be the desired result. A good general rule is to use the `scale` parameter for global scaling unless the primary objective is to specify overall dimensions.

11. **Buffer overflow:** The **m4** error message of the form `pushed back more than 4096 chars` results from expanding large macros or macro arguments, and can be avoided by enlarging the buffer. For example, the option `-B16000` enlarges the buffer size to 16000 bytes. However this error message could also result from a syntax error.

14 List of macros

The following table lists the macros in libraries darrow.m4, libcct.m4, liblog.m4, libgen.m4, and files gpic.m4, mfpic.m4, and pstricks.m4. Some of the example sources contain additional macros, such as for flowcharts and binary trees.

Internal macros defined within the libraries begin with the characters m4, and are not listed here.

The libraries define the **pic** location `M4_Tmp` and following **pic** variables:
`m4azim m4caz m4cel m4dll m4dlw m4elev m4i m4j m4saz m4sel m4t1 m4t2 m4t3 m4tmp rp_ang rp_ht rp_len rp_wid`.

The library in which each macro is found is given, and a brief description.

`AND_gate(`*n*`)`	log	basic 'and' gate, 2 or *n* inputs
`AND_ht`	log	height of basic 'and' and 'or' gates
`AND_wd`	log	width of basic 'and' and 'or' gates
`BUFFER_gate(`*linespec*`)`	log	basic buffer, 1 input or as a 2-terminal element
`BUF_ht`	log	basic buffer gate height

BUF_wd	log	basic buffer gate width
Cos(*integer*)	gen	cosine function, *integer* degrees
E_	gen	the constant *e*
Fector(x1,y1,z1,x2,y2,z2)	3D	vector projected on current view plane with top face of 3-dimensonal arrowhead normal to x2,y2,z2
FlipFlop(D|T|RS|JK)	log	experimental flip-flops
G_hht_	log	gate half-height
HOMELIB_	all	directory containing libraries
Intersect_(*Name1*,*Name2*)	gen	intersection of two named lines
L_unit	log	logic-element grid size
NAND_gate(*n*)	log	'nand' gate, 2 or *n* inputs
NOR_gate(*n*)	log	'nor' gate, 2 or *n* inputs
NOT_gate(*linespec*)	log	'not' gate, 1 input or as a 2-terminal element
NXOR_gate(*n*)	log	'nxor' gate, 2 or *n* inputs
N_diam	log	diameter of 'not' circles
OR_gate(*n*)	log	'or' gate, 2 or *n* inputs
Point_(*integer*)	gen	sets direction cosines in degrees
Rect_(*radius*,*angle*)	gen	(deg) polar-to-rectangular conversion
Sin(*integer*)	gen	sine function, *integer* degrees
XOR_gate(*n*)	log	'xor' gate, 2 or *n* inputs
above_	gen	string position above relative to current direction
abs_(*number*)	gen	absolute value function
amp(*linespec*,*size*)	cct	amplifier
arrowline(*linespec*)	cct	line (dotted, dashed permissible) with centred arrowhead
battery(*linespec*,n,R)	cct	n-cell battery: default 1 cell, R=reversed polarity
below_	gen	string position relative to current direction
bi_tr(*linespec*,L|R,P,E)	cct	left or right, N or P-type bipolar transistor, without or with envelope
boxcoord(*planar obj*,*x fraction*,*y fraction*)	gen	internal point in a planar object
boxdim(*name*,h|w|d|v,*default*)	gen	evaluate, e.g. *name*_w if defined, else *default* if given, else 0 v gives sum of d and h values
b_current(*label*,*pos*,In|Out,Start|End)	cct	draw and label branch-current arrow
capacitor(*linespec*,C,R)	cct	capacitor, C=curved-plate, R=reversed polarity
clabel(*label*,*label*,*label*)	cct	centre triple label
contact(O|C,R)	cct	single-pole contact: default double pole or normally open or closed, oriented to the left or right
consource(*linespec*,V|I)	cct	voltage or current controlled source
cosd(*arg*)	gen	cosine of an expression in degrees
cross(at *location*)	gen	plots a small cross
cross3D(x1,y1,z1,x2,y2,z2)	3D	cross product of two triples
crossover(*linespec*, L|R, Line1, ...)	cct	line jumping left or right over named lines
crosswd_	gen	cross dimension
csdim_	cct	controlled-source width
d_fet(*linespec*,L|R,P,S,E)	cct	left or right, N or P depletion MOSFET, normal or simplified, without or with envelope
darrow(*linespec*,t,t,*width*,*arrowhd wd*, *arrowhd ht*)	darrow	double arrow
dcosine3D(i,x,y,z)	3D	extract i-th entry of triple x,y,z
delay(*linespec*,*size*)	cct	delay element
delay_rad_	cct	delay radius
dend(at *location*)	darrow	close (or start) double line
diff_(*a*,*b*)	gen	difference function

diff3D(x1,y1,z1,x2,y2,z2)	3D	difference of two triples
dimen_	cct	size parameter for circuit elements
dimension_(linespec,offset,label,blank width,tic offset,arrowhead)	gen	macro for dimensioning diagrams; arrowhead=-> | <-
diode(linespec,N|B|S|T|Z|LE|LER,R)	cct	diode: normal, bi-directional, Schottky, tunnel, zener, LED with arrows, LED with arrows right, R=reversed polarity
dlabel(long,lat,label,label,label)	cct	general triple label
dleft	darrow	double line left turn
dline(linespec,t,t,width)	darrow	double line
dlinewid	darrow	width of double lines
dn_	gen	sets down relative to current-direction
dot(at location,radius,fill)	gen	draw a (filled) dot
dot3D(x1,y1,z1,x2,y2,z2)	3D	dot product of two triples
dotrad_	gen	dot radius
down_	gen	sets current direction to down
dright	darrow	double arrow right turn
dtee('direction')	darrow	double arrow tee junction
dtor_	gen	degrees to radians conversion constant
e_	gen	.e relative to current direction
e_fet(linespec,L|R,P,S,E)	cct	left or right, N or P enhancement MOSFET, normal or simplified, without or with envelope
ebox(linespec,length,ht)	cct	two-terminal box element with adjustable dimensions
eleminit_(linespec)	cct	internal line initialization
elen_	cct	default element length
em_arrows(linespec,L|R,U|D)	cct	nonionizing radiation arrows
expe	gen	exponential, base e
fill_(number)	gen	fill macro, 0=black, 1=white
fuse	cct	fuse symbol
gap(linespec,fill)	cct	gap with dots
glabel_	cct	internal general labeller
gpar_(element,element,separation)	cct	two same-direction elements in parallel
gpic_	gpic	defined to signify gpic is being used
grid_(x,y)	log	absolute grid location
ground(at location,T,N|F|S|L|P)	cct	ground, without stem for nonblank 2nd arg; N=normal, F=frame, S=signal, L=low-noise, P=protective
hop(L|R,at location)	cct	conductor crossing another to left or right
hoprad_	cct	hop radius
ht_	gen	height relative to current direction
inductor(linespec,W,n,M)	cct	inductor, narrow or wide, 4 or n arcs, without or with magnetic core
integrator(linespec,size)	cct	integrating amplifier
intersect_(line1.start,line1.end,line2.start,line2.end)	gen	intersection of two lines
j_fet(linespec,L|R,P,E)	cct	left or right, N or P JFET, without or with envelope
left_	gen	left with respect to current direction
length3D(x,y,z)	3D	Euclidean length of triple x,y,z
linethick_(number)	gen	set line thickness in points
lin_leng(line-reference)	gen	calculate the length of a line
ljust_	gen	ljust with respect to current direction
llabel(label,label,label)	cct	triple lable on left side of the element
loc_(x, y)	gen	location adjusted for current direction

log10E_	gen	constant $\log_{10}(e)$
loge	gen	logarithm, base e
lt_	gen	left with respect to current direction
manhattan	gen	sets direction cosines for left, right, up, down
mfpic_	mfpic	defined to signify mfpic is being used
mosfet(*linespec*,L|R,P,[D][B],E)	cct	MOSFET left or right, N or P, D=depletion mode, B=disconnected body, without or with envelope
m4_arrow(*linespec*,*ht*,*wid*)	gen	arrow with adjustable head, filled when possible
n_	gen	.n with respect to current direction
ne_	gen	.ne with respect to current direction
neg_	gen	unary negation
nport(*box spec,nw,nn,ne,ns,space ratio,pin lgth*)	cct	nport macro (default twoport)
nw_	gen	.nw with respect to current direction
opamp(*linespec*,*label*,*label*,*size*,P)	cct	operational amplifier with $-$, $+$ or other internal labels, specified size, and optional power connections
open_arrow(*linespec*,*ht*,*wid*)	gen	arrow with adjustable open head
par_(*element*,*element*,*separation*)	cct	two same-direction, same-length elements in parallel
point_(*angle*)	gen	(radians) set direction cosines
polar_(*x*,*y*)	gen	rectangular-to polar conversion
print3D(x,y,z)	3D	write out triple for debugging
prod_(*a*,*b*)	gen	binary multiplication
project(*x*,(*y*,(*z*)	3D	3D to 2D projection
psset_(*PSTricks settings*)	gen	set PSTricks parameters
pstricks_	pstricks	defined to signify PSTricks is being used
pt_	gen	big point size factor, in scaled inches, (*scale/72)
pt__	gen	TEX point size factor, in scaled inches, (*scale/72.27)
rect_(*radius*,*angle*)	gen	(radians) polar-rectangular conversion
relay(n,O|C,R)	cct	relay: n poles (default 1), default double throw or normally open or closed, drawn left or right of current direction
resistor(*linespec*,n)	cct	resistor, n peaks, default 3
reversed('*macro name*',args)	cct	reverse polarity of 2-terminal element
right_	gen	set current direction right
rjust_	gen	right justify with respect to current direction
rlabel(*label*,*label*,*label*)	cct	triple lable on right side of the element
rot3Dx(*radians*,x,y,z)	3D	rotates x,y,z about x axis
rot3Dy(*radians*,x,y,z)	3D	rotates x,y,z about y axis
rot3Dz(*radians*,x,y,z)	3D	rotates x,y,z about z axis
rpoint_(*linespec*)	gen	set direction cosines
rpos_(*position*)	gen	Here + *position*
rt_	gen	right with respect to current direction
rtod_	gen	constant, degrees/radian
rvec_(*x*,*y*)	gen	location relative to current direction
s_	gen	.s with respect to current direction
s_box(*text*,*expr1*,$\cdots$)	gen	generate dimensioned text string using \boxdims from boxdims.sty. Two or more args are passed to sprintf().
s_init(*name*)	gen	initialize s_box string label to *name* which should be unique
scr(*linespec*,R,G,E)	cct	triac (scr), right, gated, envelope
se_	gen	.se with respect to current direction
setview(*azimuth degrees*,*elevation degrees*)	3D	set projection viewpoint
sfg_init(*default line len, node rad, arrowhd len, arrowhd wid*)	cct	initialization of signal flow graph macros
sfgabove	cct	like above but with extra space
sfgbelow	cct	like below but with extra space

`sfgarc(`*linespec*`,`*text*`,`*text justification*`,cw|ccw,`*height scale factor*`)`		
	cct	directed arc drawn between nodes, with text label and a height-adjustment parameter
`sfgline(`*linespec*`,`*text*`,`*text justification*`)`		
	cct	directed straight line chopped by node radius, with text label
`sfgnode(at` *location*`,`*text*`,above|below)`		
	cct	white small circle, with text label
`shade(`*gray value*`,`*closed line specs*`)`		
	gen	fill arbitrary closed curve
`shadebox(`*box specification*`)`	gen	box with edge shading
`sign_(`*number*`)`	gen	sign function
`sind(`*arg*`)`	gen	sine of an expression in degrees
`smosfet(`*linespec*`,L|R,P,D,E)`	cct	simplified MOSFET left or right, N or P, enhancement or depletion, without or with envelope
`source(`*linespec*`,V|I|AC|X|`*string*`,`*diameter*`)`		
	cct	source, blank or voltage or current or AC or X or labelled
`sourcerad_`	cct	default source radius
`sprod3D(a,x,y,z)`	3D	scalar product of triple x,y,z by a
`sp_`	gen	evaluates to medium space for gpic strings
`sum_(`a,b`)`	gen	binary sum
`sum3D(x1,y1,z1,x2,y2,z2)`	3D	sum of two triples
`svec_(`x,y`)`	log	scaled and rotated grid coordinate vector
`sw_`	gen	.sw with respect to current direction
`switch(`*linespec*`,L|R,C|O,B)`	cct	SPST switch left or right, blank or closing or opening arrow, or button
`thicklines_(`*number*`)`	gen	set line thickness in points
`thinlines_(`*number*`)`	gen	set line thickness in points
`transformer(`*linespec*`,L|R,np,A,ns)`		
	cct	2-winding transformer: left or right, np primary arcs, air core, ns secondary arcs
`twopi_`	gen	2π
`ujt(`*linespec*`,R,P,E)`	cct	unijunction transistor, right, P-channel, envelope
`unit3D(x,y,z)`	3D	unit triple in the direction of triple x,y,z
`up_`	gen	set current direction up
`up__`	gen	up with respect to current direction
`vec_(`x,y`)`	gen	position rotated with respect to current direction
`vrot_(`$x,y,$*xcosine*`,`*ycosine*`)`	gen	rotation operator
`vlength(`x,y`)`	gen	vector length $\sqrt{x^2+y^2}$
`vscal_(`*number*`,`x,y`)`	gen	vector scale operator
`w_`	gen	.w with respect to current direction
`wid_`	gen	width with respect to current direction
`xtal(`*linespec*`)`	cct	quartz crystal

References

[1] J. Bentley. *More Programming Pearls.* Addison-Wesley, Reading, Massachusetts, 1988.

[2] A. R. Clark. Using circuit macros, 1999. Courtesy of Alan Robert Clark at `http://ytdp.ee.wits.ac.za/cct.html`.

[3] The Free Software Foundation. Gpic man page, 1992.

[4] M. Goossens, S. Rahtz, and F. Mittelbach. *The LaTeX Graphics Companion.* Addison-Wesley, Reading, Massachusetts, 1997.

[5] J. D. Hobby. A user's manual for MetaPost, 1990.

[6] IEEE. Graphic symbols for electrical and electronic diagrams, 1975. Std 315-1975, 315A-1986, reaffirmed 1993.

[7] B. W. Kernighan and D. M. Richie. The M4 macro processor. Technical report, Bell Laboratories, 1977.

[8] B. W. Kernighan and D. M. Richie. PIC—A graphics language for typesetting, user manual. Technical Report 116, AT&T Bell Laboratories, 1991.

[9] Thomas K. Landauer. *The Trouble with Computers*. MIT Press, Cambridge, 1995.

[10] E. S. Raymond. Making pictures with GNU PIC, 1995. In GNU groff source distribution.

[11] T. Rokicki. DVIPS: A TeX driver. Technical report, Stanford, 1994.

[12] A. S. Sedra and K. C. Smith. *Microelectronic Circuits*. Oxford University Press, Oxford, 1997.

[13] R. Seindal. GNU m4, version 1.4, 1994. `http://www.gnu.org/manual/m4-1.4/html_mono/m4.html`.

[14] T. Van Zandt. PSTricks user's guide, 1993.