M4 Macros for Electric Circuit Diagrams in LaTeX Documents

Dwight Aplevich

Version 6.9

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 [10].

This document describes a set of macros, written in the m4 macro language [8], for producing electric circuits and other diagrams in LaTeX documents. The macros evaluate to drawing commands in pic, a line-drawing language [9] that is readily available and 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 can be 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. It is often convenient to customize elements or to package combinations of them for particular drawings, so macros such as these are only a starting point.

2 Using the macros

This section describes the basic process of adding circuit diagrams to LaTeX documents to produce postscript or pdf files. On some operating systems, project management software with graphical interfaces can be used to automate the process but, for simple documents, the steps can be performed by hand as described in Section 2.1.

The diagram source file is preprocessed as illustrated in Figure 1. The predefined macros, followed by the diagram source, are read by m4. The result is passed through a pic interpreter to produce `.tex` output that can be inserted into a `.tex` document using the `\input` command.

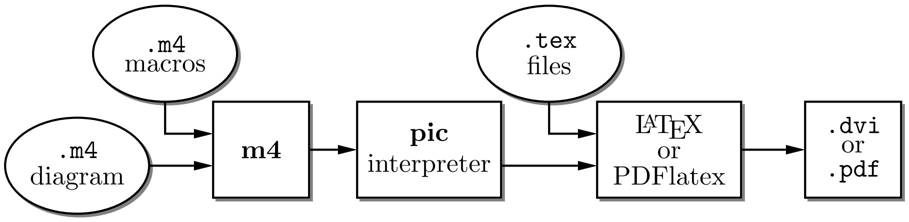

Figure 1: Inclusion of figures and macros in the LaTeX document. Replacing LaTeX with PDFlatex to produce pdf directly is also possible.

Depending on the pic interpreter chosen, the choice of options, and the choice of LaTeX or PDFlatex, the interpreter output may contain tpic specials, LaTeX graphics, PSTricks [14] commands, Ti*k*z PGF commands, or other formats. These variations are described in Section 12.

There are two principal choices of pic interpreter. One is dpic, described later in this document. An alternative [3] is gpic -t together with a printer driver that understands tpic specials, typically [12] 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. Pic processors contain basic macro facilities, so some of the concepts applied here require only a pic processor. By judicious use of macros, features of both m4 and pic can be exploited.

2.1 Quick start

The contents of file `quick.m4` and resulting diagram are shown in Figure 2 to illustrate the language, to show several ways for placing circuit elements, and to provide sufficient information for producing basic labeled circuits.

To process the file, make sure that the libraries `libcct.m4` and `libgen.m4` are installed and readable. Verify that m4 is installed. Now there are at least two possibilities, described in the following subsections, with somewhat simpler usage to be given in Section 2.1.3.

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

elen = 0.75                    # Variables are allowed; default units are inches
Origin: Here                   # Position names are capitalized
   source(up_ elen); llabel(-,v_s,+)
   resistor(right_ elen);  rlabel(,R,)
   dot
   {                           # 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 right_ elen*2/3
   inductor(down_ Here.y-Origin.y); rlabel(,L,); b_current(i)
   line to Origin
.PE                            # Pic input ends
```

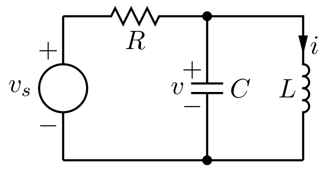

Figure 2: The file `quick.m4` and resulting diagram.

2.1.1 Processing with dpic and PSTricks or Ti*k*z PGF

If you are using dpic with PSTricks, type the following commands or put them into a script:

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

where `<path>` is the path to the `libcct.m4` file. Put `\usepackage{pstricks}` in the main LaTeX source file header and the following in the body:

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

This distribution is compatible with the Ti*k*z PGF drawing commands, which have nearly the power of the PSTricks package with the ability to produce `pdf` output by running the `pdflatex` command instead of `latex` on the input file. The commands are modified to read `pgf.m4` and invoke the dpic `-g` option as follows:

```
m4 <path>pgf.m4 <path>libcct.m4 quick.m4 > quick.pic
dpic -g quick.pic > quick.tex
```

The LaTeX header should contain `\usepackage{tikz}`, but the inclusion statemensts are the same as for PSTricks input.

In all cases the essential line is `\input quick`, which inserts the previously created file `quick.tex`. Then LaTeX the document, convert to postscript typically using dvips, and print the result or view it using Ghostview. The alternative for Ti*k*z PGF output of `dpic -g` is to invoke PDFlatex.

2.1.2 Processing with gpic

If your printer driver understands tpic specials and you are using gpic (on some systems the gpic command is `pic`), the commands are

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

The figure inclusion statements are as shown:

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

2.1.3 Simplifications

- These macros can be processed by LaTeX-specific project software and by graphic applications such as Cirkuit[7].
- 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 or just after the `.PS` line:

  ```
  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 employed to automate the commands given above. On systems without such a facility, a scripting language can be used.
- You can put several diagrams into a single source file so that they can be 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.
- It may be desirable to invoke m4 and dpic automatically from the document file, as shown:

```
\documentclass{article}
\usepackage{tikz}
\newcommand\mtopgf[1]{\immediate\write18{m4 <path>/pgf.m4 #1.m4 | dpic -g > #1.tex}}%
\begin{document}
\mtopgf{PicA}
\input{PicA.tex} \par
\mtopgf{PicB}
\input{PicB.tex}
\end{document}
```

The path to `pgf.m4` must be defined in this file, sources `PicA.m4` and `PicB.m4` must contain any required `include` statements, and the document file should be processed using the command `pdflatex -shell-escape` <filename>. This method processes each picture source every time LaTeX is run, so for large documents containing many graphics, the `\mtopgf` line could be commented out after debugging the corresponding graphic.

3 Pic essentials

Pic source is a sequence of lines in a file. The first line of a diagram begins with `.PS` with optional following arguments, and the last line is normally `.PE`. Lines outside of these pass 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 consists of a pair of square brackets enclosing other objects, as described in Section 3.4. Objects can be placed using absolute coordinates or relative to other objects.

Pic allows the definition of real-valued variables, which are alphameric names beginning with lower-case letters, and computations using them. Objects or locations on the diagram can be given symbolic names beginning with an upper-case first letter.

3.1 Manuals

At the time of writing, the classic pic manual [9] can be obtained from URL:
`http://www.cs.bell-labs.com/10thEdMan/pic.pdf`

A more complete manual [11] is included in the GNU groff package. A compressed postscript version is available, at least temporarily, with these circuit files.

In both of the above manuals, 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 "little languages" for document production, and of pic in particular, see Chapter 9 of [1]. Chapter 1 of [5] also contains a brief discussion of this and other languages.

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 construction

```
line from A to B chop x
```

truncates the line at each end by x or, if x is omitted, by the current circle radius, which is convenent when A and B are symbolic names for circular graph nodes, for example. Otherwise

```
line from A to B chop x chop y
```

truncates the line ends by x and y, which may be negative.

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

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 and is equal to $(0, 0)$ at the beginning of a diagram or block. 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, as shown in the following example, which also illustrates how to refer to a previously drawn element if it has not been given a name:

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

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. Note that

```
arc cw from Bus23.start to Bus23.end
```

does *not* define the arc uniquely; there are two arcs that satisfy this specification. This distribution includes the m4 macros

`arcr(` *position, radius, start radians, end radians*)
`arcd(` *position, radius, start degrees, end degrees*)
`arca(` *chord linespec,* `ccw|cw,` *radius, modifiers*)

to draw uniquely defined arcs. For example,

```
arcd((1,1),2,0,-90) -> dashed cw
```

draws a clockwise arc with centre at $(1, 1)$, radius 2, from $(3, 1)$ to $(1, -1)$, and

```
arca(from (1,1) to (2,2),,1,->)
```

draws an acute-angled arc with arrowhead on the chord defined by the first argument.

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,` and text

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

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

after which, in this example, the position `A.center` is defined, and can be referenced 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 automatically 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 or colour; thus

```
box dashed fill
```

produces a dashed box filled with a medium gray by default. The gray density can be controlled using the `fill_(`*number*`)` macro, where $0 \leq number \leq 1$, with 0 corresponding to black and 1 to white.

Basic colours for lines and fills are provided by gpic and dpic, but more elaborate line and fill styles 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 inserted.

Arbitrary text strings, typically meant to be typeset by LaTeX, are delimited by double-quote characters and occur in two ways. The first way is illustrated by

`"\large Resonances of $C_{20}H_{42}$" wid` *x* `ht` *y* `at` *position*

which writes the typeset result, like a box, at *position* and tells pic its size. The default size assumed by pic is given by parameters `textwid` and `textht` if it is not specified as above. The exact typeset size of formatted text can be obtained as described in Section 10. The second occurrence associates one or more strings with an object, e.g., the following writes two words, one above the other, at the centre of an ellipse:

```
ellipse "\bf Stop" "\bf here"
```

The C-like pic function `sprintf("`*format string*`",`*numerical arguments*`)` is equivalent to a string.

3.4 Compound objects

A compound object is a group of statements enclosed in square brackets. Such an object is placed by default as if it were a box, but it can also be placed by specifying the final position of an internal location. Consider the example code fragment shown:

```
Ands: [ right_
        And1: AND_gate
        And2: AND_gate at And1 - (0,And1.ht*3/2)
        line from And1.Out right_ And1.wid/3 then down_ (And1.y-And2.y)/2 then \
          left_ And1.wid*5/3 then to And2.In1-(And1.wid/3,0) then to And2.In1
      ...
      ] with .And2.In1 at (K.x,IC5.Pin9.y)
```

The two gate macros evaluate to compound objects containing `Out`, `In1`, and other locations. The final positions of all objects between the square brackets are specified in the last line by specifying the position of `In1` of gate `And2`.

3.5 Other language facilities

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`, variables, horizontal or vertical coordinates of terms such as *position*`.x` or *position*`.y`, dimensions of pic objects, e.g. `last circle.rad`. The elementary algebraic operators are `+, -, *, /, %, =, +=, -=, *=, /=,` and `%=,` similar to the C language.

The logical operators `==, !=, <=, >=, >,` and `<` 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 two-argument functions `atan2, max, min,` and the random-number generator `rand()`. Other functions are also provided using macros.

A pic manual should be consulted for details, more examples, and 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, pic macros, and external file inclusion.

4 Two-terminal circuit elements

There is a fundamental difference between two-terminal elements, which are drawn as directed linear objects, and other elements, which are compound objects as described in Section 3.4. The two-terminal element macros follow a set of conventions described in this section, and other elements will be described in Section 5.

4.1 Circuit and element basics

First, the arguments of all drawing macros have default values, so that only arguments that differ from these values need be specified. The arguments are given in Section 15.

Consider the resistor shown in Figure 3, which also serves as an example of pic commands. The first part of the source file for this figure is as follows:

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

R1: resistor
```

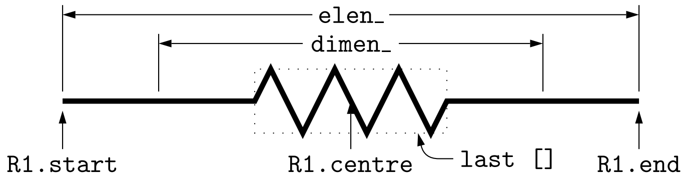

Figure 3: Resistor named `R1`, showing the size parameters, enclosing block, and predefined positions.

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

- The first line invokes an almost-empty macro `cct_init` that initializes local variables needed 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.
- The body dimensions of two-terminal elements are multiples of the macro `dimen_`, which evaluates by default to `linewid`, the pic environmental parameter with default value 0.5 in. The default length of an element is `elen_`, which is `dimen_*3/2`. Setting `linewid` to 2.0 as in the example means that the default element length becomes 3.0 in. For resistors, the length of the body is `dimen_/2,` and the width is `dimen_/6.` All of these values can be customized. Element scaling is discussed further in Section 8.

- The macro `linethick_` sets the thickness of subsequent lines (to 2.0 pt in the example). In the m4 language, macro arguments are written within parentheses following the macro name, with no space between the name and the opening parenthesis. Lines can be broken before a macro argument because m4 ignores white space immediately preceding arguments.
- 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 the line is drawn a distance `elen_` in the current direction, which is to the right by default. All this is handled by the macro `eleminit_`, which also calculates the length and angle of the invisible line for later use. The invisible line is first drawn, then the element is drawn on top of it. The element—rather the initial invisible line—can be given a name, `R1` in the example, so that positions `R1.start`, `R1.centre`, and `R1.end` are defined as shown.
- The element body is enclosed by a block, which 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. A dotted box has been drawn in the diagram to show the block boundaries.
- The last sub-element, identical to the first in two-terminal elements, is an invisible line that can be referenced later to place labels or other elements. This might be over-kill. If you create your own macros you might choose simplicity over generality, and only include visible lines.

To produce Figure 3, the following embellishments were included after the previously shown source:

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

   move to 0.85<last [].sw,last [].se>
   spline <- down arrowht*2 right arrowht/2 then right 0.15; "\tt last []" ljust

   arrow <- down 0.3 from R1.start chop 0.05; "\tt R1.start" below
   arrow <- down 0.3 from R1.end chop 0.05; "\tt R1.end" below
   arrow <- down last [].c.y-last arrow.end.y from R1.c; "\tt R1.centre" 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 0.4 pt, 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.

4.2 The two-terminal elements

Figures 4–9 are tables of the two-terminal elements. Several elements are included more than once to illustrate some of their arguments, which are listed in Section 15.

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

```
   resistor(up_ 1.25,7)
```

draws a resistor 1.25 units long up from the current position, with 7 vertices per side. The macro `up_` evaluates to `up` but also resets the current directional parameters to point up.

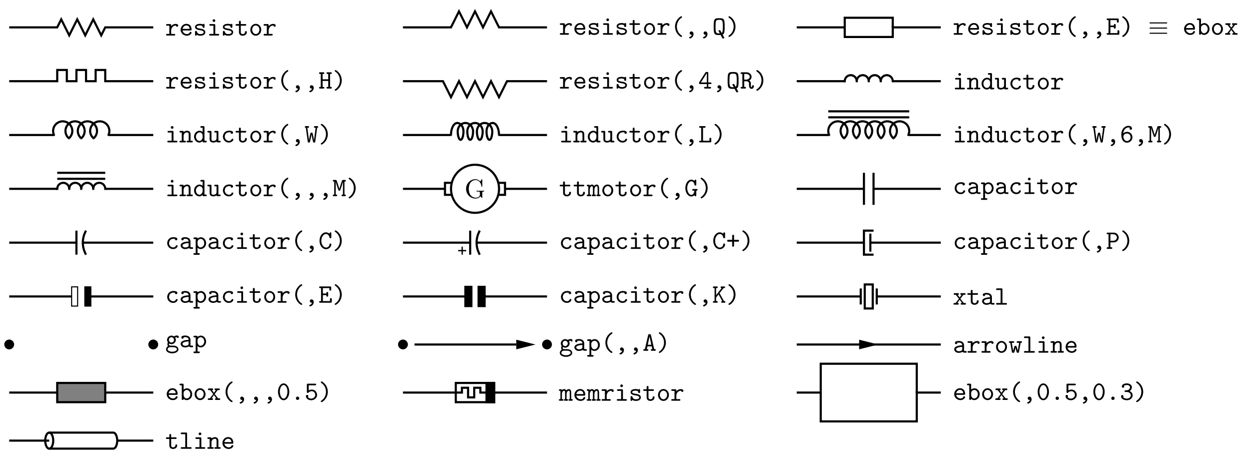

Figure 4: Two-terminal elements, showing some variations.

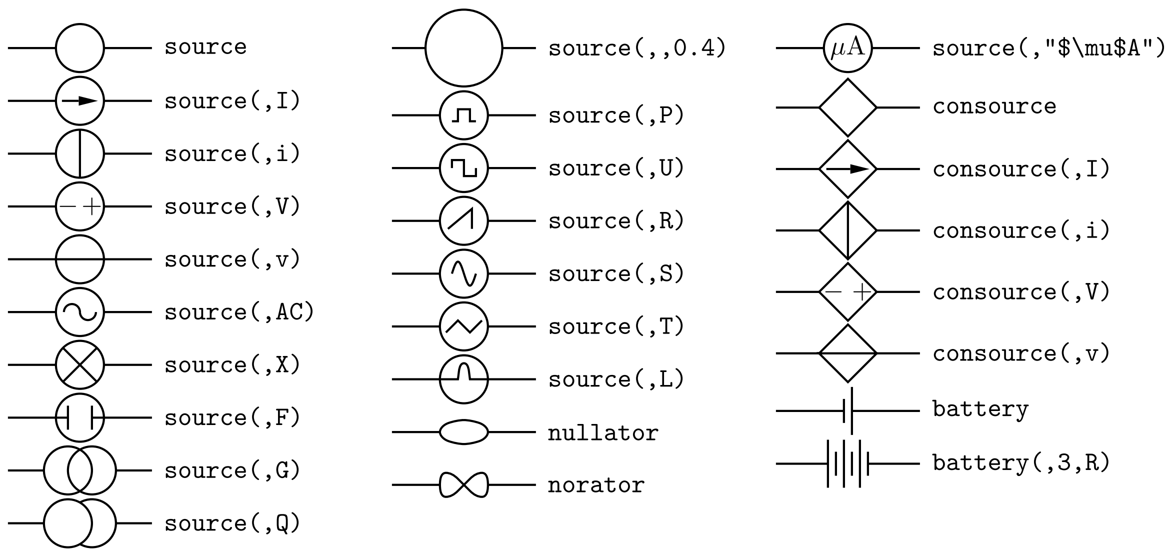

Figure 5: Sources and source-like elements.

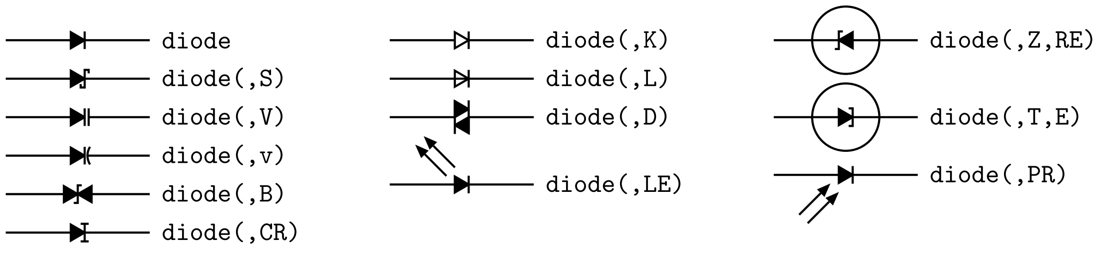

Figure 6: Variants of `diode(`*linespec*`,B|D|K|L|LE[R]|P[R]|S|T|V|Z,[R][E])`.

Most of the two-terminal elements are oriented; that is, they have a defined polarity. Several element macros include 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)`

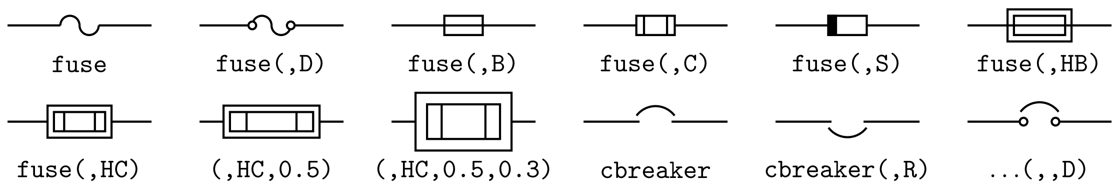

Figure 7: The `fuse(`*linespec*`, A|dA|B|C|D|E|S|HB|HC, `*wid*`, `*ht*`)` and `cbreaker(`*linespec*`,L|R,D)` macros.

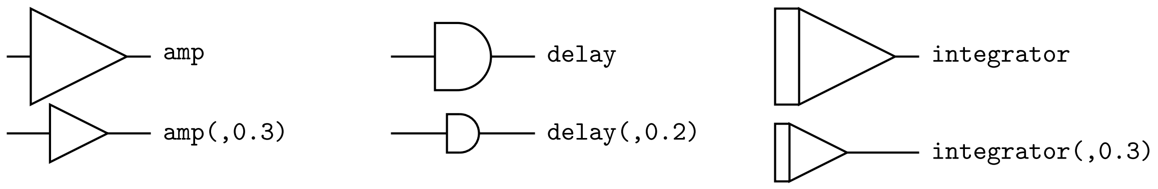

Figure 8: Amplifier, delay, and integrator.

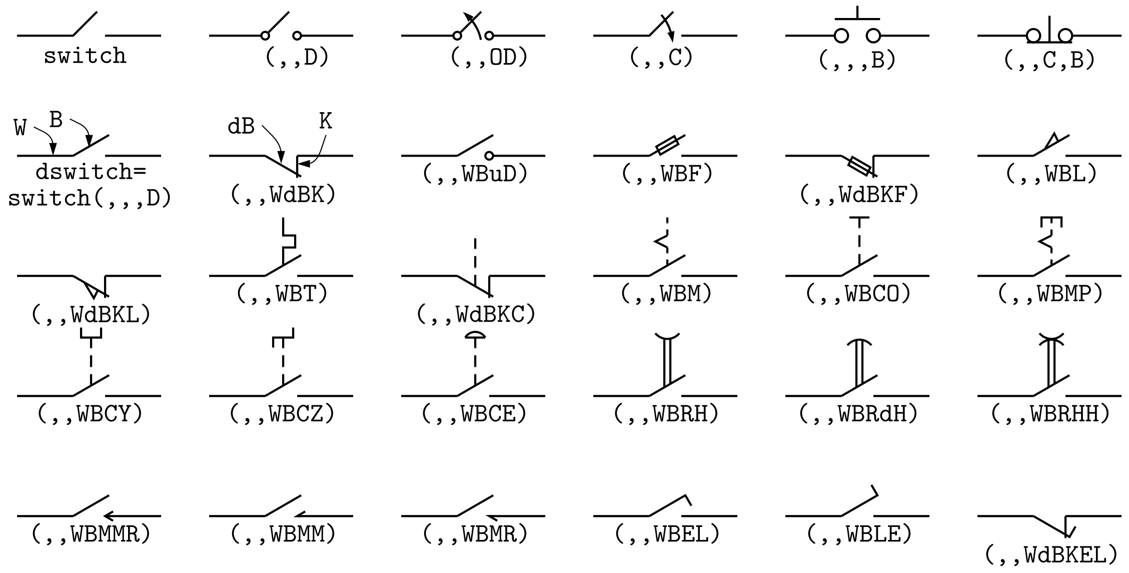

Figure 9: The basic `switch(`*linespec*`,L|R,[O|C][D],B)` macro and more elabourate `dswitch(`*linespec*`,R,W[ud]B[K]`*chars*`)` macro, with drawing direction `right_`. Setting the second argument to `R` produces a mirror image with respect to the drawing direction. The macro `switch(,,,D)` is a wrapper for the comprehensive `dswitch` macro.

draws two diodes to the right, but the second one points left.

Figure 10 shows some two-terminal elements with arrows or lines overlaid to indicate variability using the macro `variable(`‘*element*’`,`*type*`,`*angle*`,`*length*`)`, where *type* is one of `A, P, L, N,` with `C` or `S` optionally appended to indicate continuous or stepwise variation. Alternatively, this macro can be invoked similarly to the label macros in Section 4.4 by specifying an empty first argument; thus, the following line draws the resistor in Figure 10:

`resistor(down_ dimen_); variable(,uN)`

Figure 11 contains arrows for indicating radiation effects. The arrow stems are named *A1*, *A2*, and each pair is drawn in a `[]` block, with the names *Head* and *Tail* defined to aid placement near another device. The second argument specifies absolute angle in degrees (default 135 degrees).

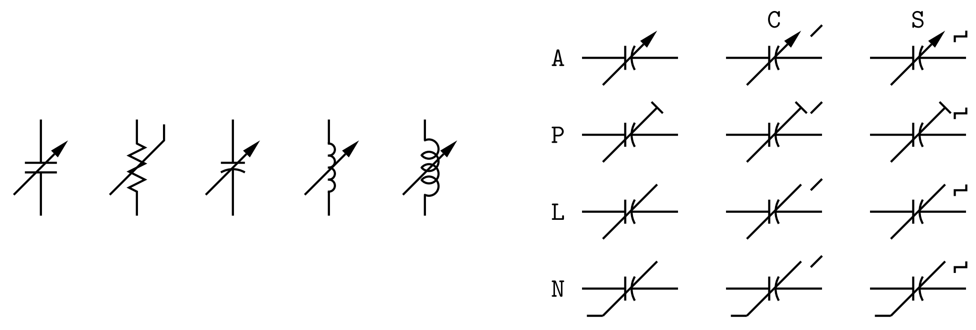

Figure 10: Illustrating `variable(`‘*element*’`,[A|P|L|[u]N][C|S],`*angle*`,`*length*`)`. For example, `variable(‘capacitor(down_ dimen_)’)` draws the leftmost capacitor shown above, and `variable(‘resistor(down_ dimen_)’,uN)` draws the resistor. The default angle is 45°, regardless of the direction of the element. The array on the right shows the effect of the second argument.

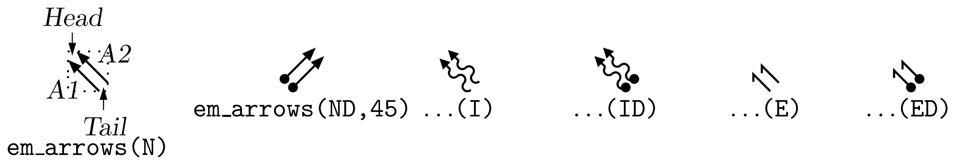

Figure 11: Radiation arrows: `em_arrows(`*type*, *angle*, *length*`)`

4.3 Branch-current arrows

Arrowheads and labels can be added to conductors using basic pic statements. For example, the following line adds a labeled arrowhead at a distance `alpha` along a horizontal line that has just been drawn. Many variations of this are possible:

```
arrow right arrowht from last line.start+(alpha,0) "$i_1$" above
```

Macros have been defined to simplify the labelling of two-terminal elements. The macro

`b_current(`*label*, `above_|below_, In|O[ut], Start|E[nd],` *frac*`)`

draws an arrow from the start of the last-drawn two-terminal element *frac* of the way toward the body. If the fourth argument is `End`, the arrow is drawn from the end toward the body. If the third element is `Out`, the arrow is drawn outward from the body. The first argument is the desired 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. The label is assumed to be in math mode unless it begins with `sprintf` or a double quote, in which case it is copied literally. A non-blank second argument specifies the relative position of the label with respect to the arrow, for example `below_`, which places the label below with respect to the current direction. Absolute positions, for example `below` or `ljust`, also can be specified. Figure 12 illustrates the resulting eight possibilities.

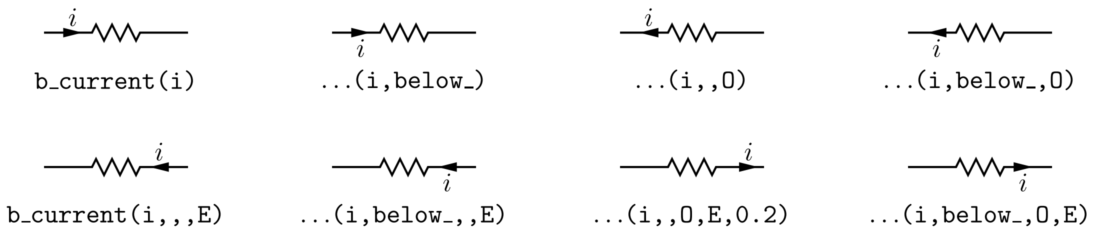

Figure 12: Illustrating `b_current`. In all cases the drawing direction is to the right.

For those who prefer a separate arrow to indicate the reference direction for current, the macros `larrow(`*label*`, ->|<-,`*dist*`)` and `rarrow(`*label*`, ->|<-,`*dist*`)` are provided. The label is placed outside the arrow as shown in Figure 13. The first argument is assumed to be in math mode unless it begins with `sprintf` or a double quote, in which case the argument is copied literally. The third argument specifies the separation from the element.

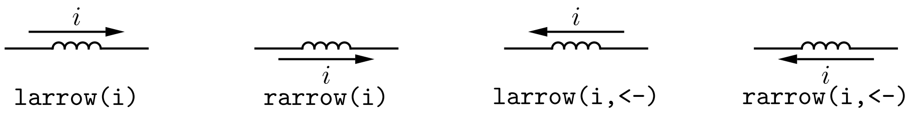

Figure 13: The `larrow` and `rarrow` macros draw reference-direction arrows adjacent to the element.

4.4 Labels

Macros for labeling 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 shown in Figure 14. The macro `dlabel` performs these 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 `sprintf` or a double quote are copied literally rather than assumed to be in math mode.

```
% ‘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
```

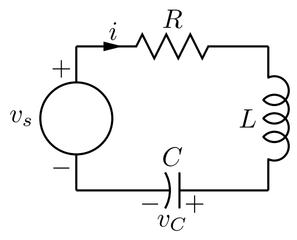

Figure 14: A loop containing labeled elements, with its source code.

5 Other circuit elements

Many basic elements are not two-terminal. These elements are usually enclosed in a block, and contain named locations in the interior. In some cases, 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 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*. A few macros are positioned with the first argument; the ground macro, for example: *element*`(at` *position*`)`.

The macro `potentiometer(`*linespec*`,`*cycles*`,`*fractional pos*`,`*length*`,...)`, shown in Figure 15, first draws a resistor along the specified line, then adds arrows for taps at fractional positions along the body, with default or specified length. A negative length draws the arrow from the right of the current drawing direction.

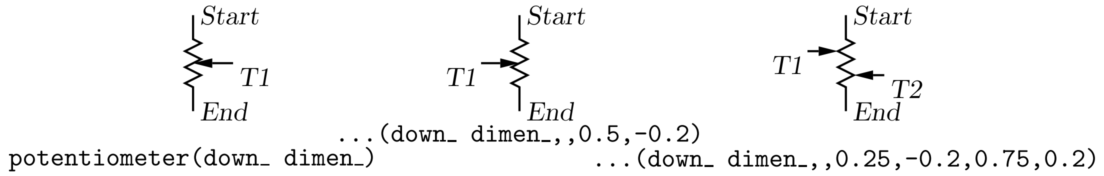
Start
T1
End
Start
T1
End
Start
T1
T2
End
...(down_ dimen_,,0.5,-0.2)
potentiometer(down_ dimen_)
...(down_ dimen_,,0.25,-0.2,0.75,0.2)

Figure 15: Default and multiple-tap potentiometer.

The ground symbol is shown in Figure 16. The first argument specifies position; for example, the two lines shown have identical effect:

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

The second argument truncates the stem, and the third defines the symbol type. The fourth argument specifies the angle at which the symbol is drawn, with down the default.

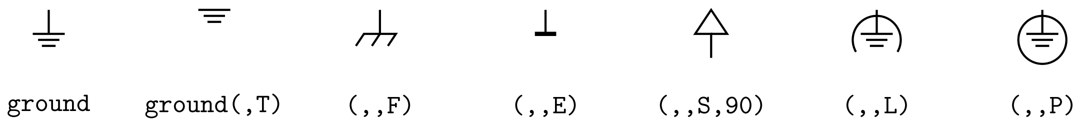

Figure 16: The `ground( at` *position*`, T, N|F|S|L|P|E, U|D|L|R|angle )` macro.

The arguments of the macro `antenna( at` *position*`, T, A|L|T|S|D|P|F, U|D|L|R|angle )` shown in Figure 17 are similar to those of `ground`.

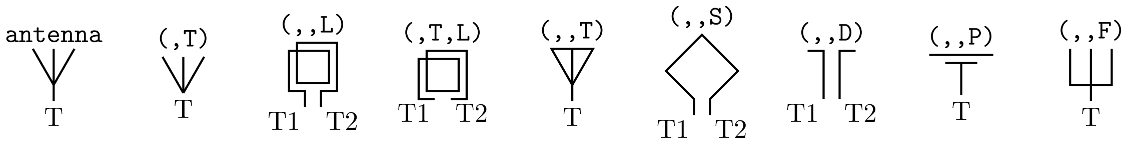

Figure 17: Antenna symbols, with macro arguments shown above and predefined terminal names below.

Figure 18 illustrates the macro `opamp(`*linespec, - label, + label, size,* *chars*`,` *other commands*`)`. The element is enclosed in a block containing the predefined internal locations shown. These locations 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 the 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 `\scriptsize$-$` and `\scriptsize$+$` respectively, and the fourth argument changes the size of the opamp. The fifth argument is a string of characters. `P` adds a power connection, `R` exchanges the second and third entries, and `T` truncates the opamp point. Customizations can be added using the last argument, which is executed within the `[ ]` block after all other elements are drawn.

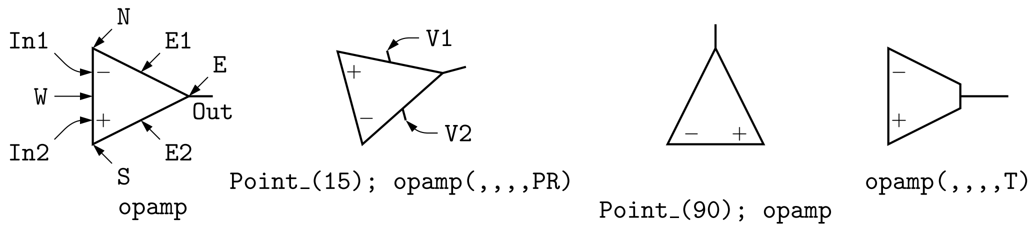

Figure 18: Operational amplifiers. The `P` option adds power connections. The second and third arguments can be used to place and rotate arbitrary text at `In1` and `In2`.

Typeset text associated with circuit elements is not rotated by default, as illustrated by the second and third opamps in Figure 18. The `opamp` labels can be rotated if necessary by using postprocessor commands (for example PSTricks `\rput`) as second and third arguments.

The code in Figure 19 places an opamp with three connections.

Figure 20 shows variants of the transformer macro, which has predefined internal locations *P1, P2, S1, S2, TP,* and *TS.* 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 and fifth arguments specifies the number of primary and secondary arcs respectively. If the fourth

```
line right 0.2 then up 0.1
A: opamp(up_,,,0.4,R) with .In1 at Here
  line right 0.2 from A.Out
  line down 0.1 from A.In2 then right 0.2
```

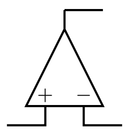

Figure 19: A code fragment invoking the `opamp(`*linespec*`,-,+,`*size*`,[R][P])` macro.

argument string contains an `A`, the iron core is omitted, and if it contains a `W`, wide windings are drawn.

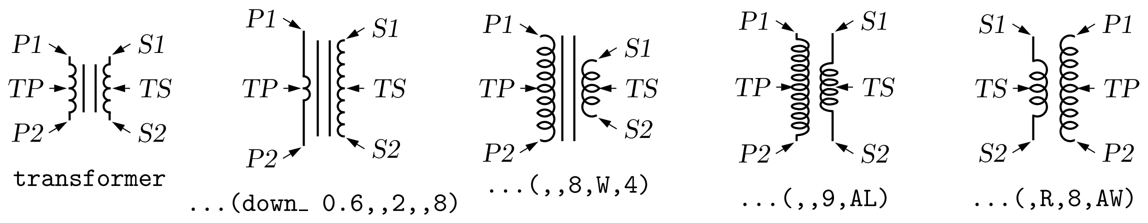

Figure 20: The `transformer(`*linespec*`,L|R,`*np*`,[A][W|L],`*ns*`)` macro (drawing direction `down`), showing predefined terminal and centre-tap points.

Figure 21 shows some audio devices, defined in `[]` blocks, with predefined internal locations as shown. The first argument specifies the device orientation. Thus,

`S: speaker(U) with .In2 at Here`

places an upward-facing speaker with input *In2* at the current location.

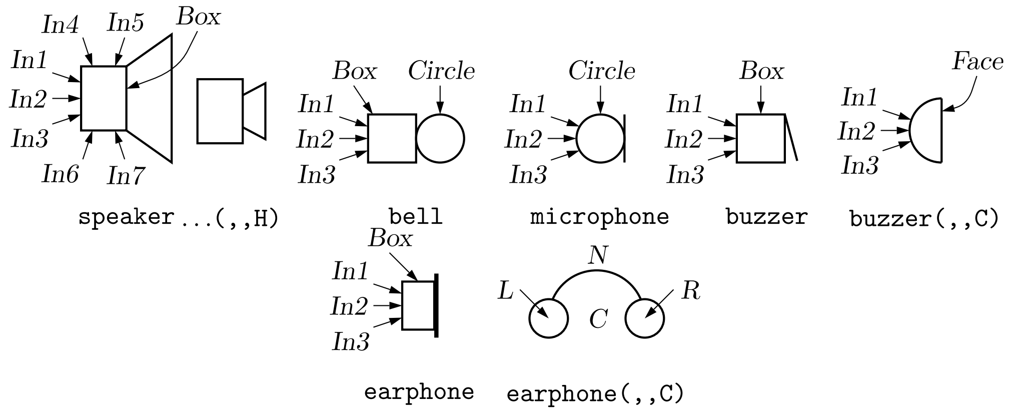

Figure 21: Audio components: `speaker(U|D|L|R|`*degrees*`,`*size*`,`*type*`)`, `bell`, `microphone`, `buzzer`, `earphone`, with their internally named positions and components.

The `nport(`*box specs, nw, nn, ne, ns, space ratio, pin lgth, style*`)` macro is shown in Figure 22. 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, the seventh is the pin length, and setting the eighth argument to `N` omits the pin dots. The complete structure is enclosed in a block.

The `nterm(`*box specs, nw, nn, ne, ns, pin lgth, style*`)` macro illustrated in Figure 22 is similar to the `nport` macro but has one fewer argument, draws single pins instead of pin pairs, and defaults to a 3-terminal box.

Many custom labels or added elements may be required, particularly for 2-ports. These elements can be added using the first argument and the ninth (not mentioned above) of the `nport` macro. For example, the following code adds a pair of labels to the box immediately after drawing it but within the enclosing block:

`nport(; ‘"0"’ at Box.w ljust; ‘"∞"’ at Box.e rjust)`

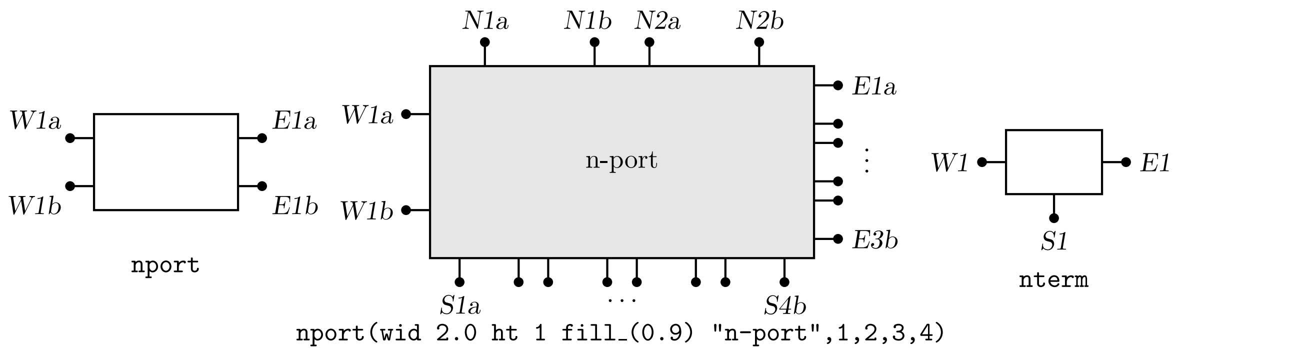

Figure 22: The `nport` macro draws a sequence of pairs of named pins on each side of a box. The pin names are shown. The default is a twoport. The `nterm` macro draws single pins instead of pin pairs.

If this were to be used extensively, then the following custom wrapper would save typing, add the labels, and pass all arguments to `nport`:

```
define(`nullor',`nport(`$1'
  {`"${}0$"' at Box.w ljust
   `"$\infty$"' at Box.e rjust},shift($@))')
```

The above example and the related gyrator macro are illustrated in Figure 23.

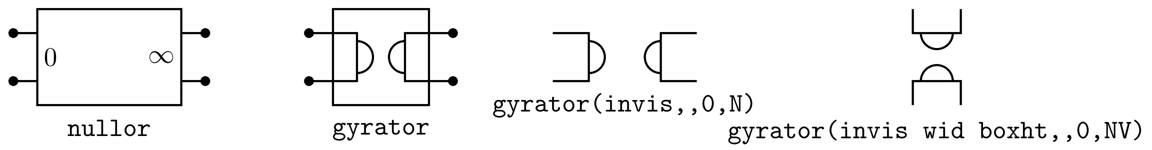

Figure 23: The `nullor` example and the `gyrator` macro are customizations of the `nport` macro.

A basic winding macro for magnetic-circuit sketches and similar figures is shown in Figure 24. For simplicity, the complete spline is first drawn and then blanked in appropriate places using the background (core) color (`lightgray` for example, default `white`).

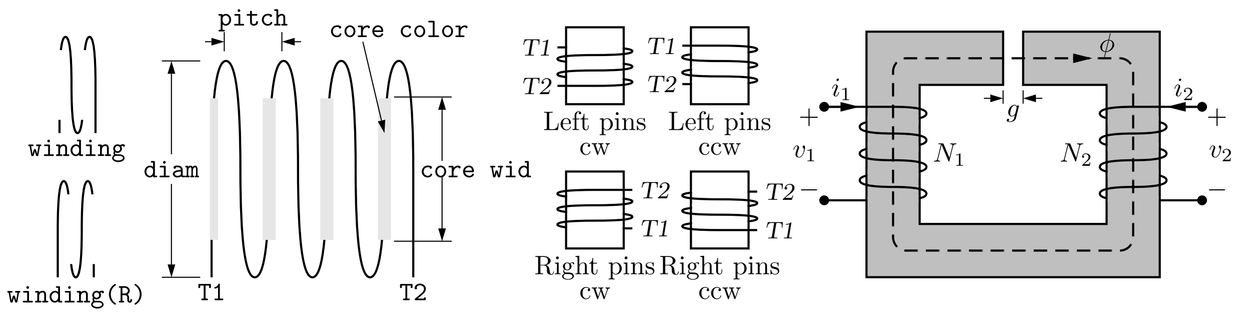

Figure 24: The `winding(L|R, diam, pitch, turns, core wid, core color)` macro draws a coil with axis along the current drawing direction. Terminals `T1` and `T2` are defined. Setting the first argument to `R` draws a right-hand winding.

Figure 25 shows 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.

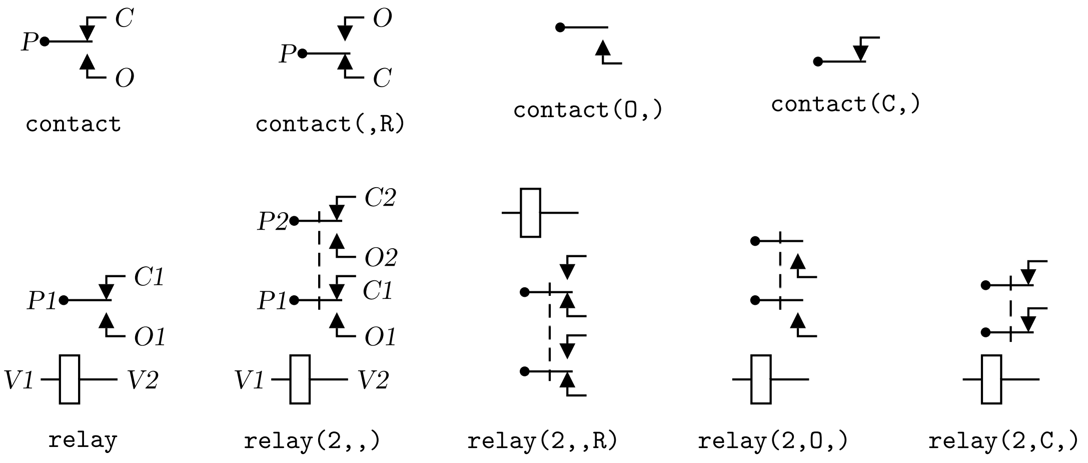

Figure 25: The `contact(O|C,R)` and `relay(`*poles*`,O|C,R)` macros (default direction right).

Figure 26 shows the variants of bipolar transistor macro
`bi_tr(`*linespec*`,L|R,P,E)`
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.

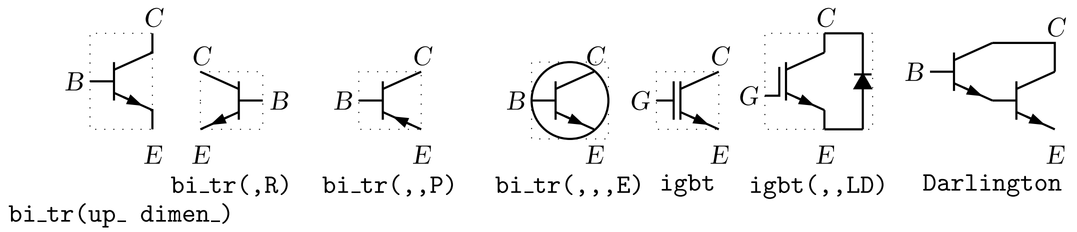

Figure 26: Bipolar transistor variants (current direction upward).

The code fragment example in Figure 27 places a bipolar transistor, connects a ground to the emitter, and connects a resistor to the collector.

```
S: dot; line left_ 0.1; up_
Q1: bi_tr(,R) with .B at Here
ground(at Q1.E)
line up 0.1 from Q1.C; resistor(right_ S.x-Here.x); dot
```

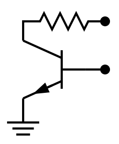

Figure 27: The `bi_tr(`*linespec*`,L|R,P,E)` macro.

The `bi_tr` and `igbt` macros are wrappers for the macro `bi_trans(`*linespec*`, L|R,` *chars*`, E)`, which draws the components of the transistor according to the characters in its third argument. For example, multiple emitters and collectors can be specified as shown in Figure 28.

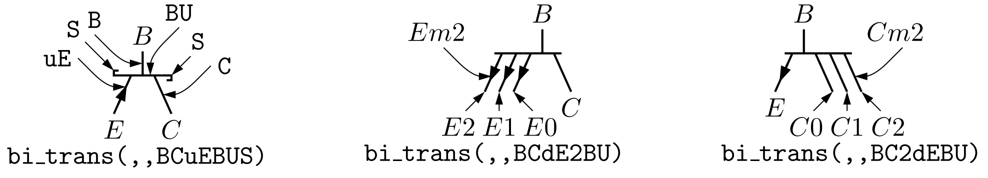

Figure 28: The `bi_trans(`*linespec*`,L|R,`*chars*`,E)` macro. The sub-elements are specified by the third argument. The substring `E`*n* creates multiple emitters *E0* to *En*. Collectors are similar.

A UJT macro with predefined internal locations *B1*, *B2*, and *E* is illustrated in Figure 29, and a thyristor macro with predefined internal locations *T1*, *T2*, and *G* is illustrated in Figure 30.

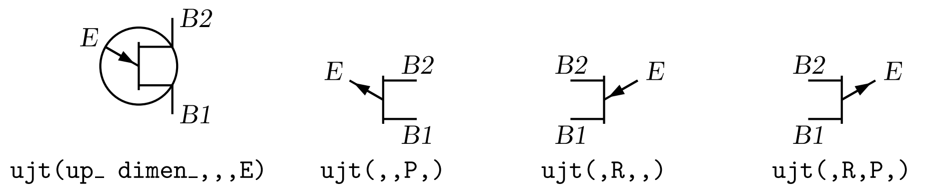

Figure 29: UJT devices, with current drawing direction up.

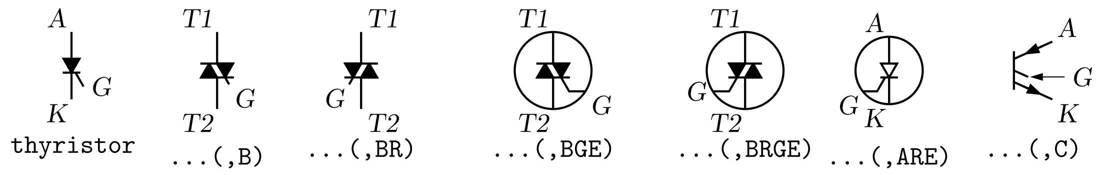

Figure 30: The `thyristor` macro, drawing direction down. These are not two-terminal elements, so the *linespec* argument determines direction and length but not position.

Some FETs with predefined internal locations *S*, *D*, and *G* are also included, with similar arguments to those of `bi_tr`, as shown in Figure 31. In all cases the first argument is a linespec, and entering `R` as the second argument orients the *G* terminal to the right of the current drawing direction. The macros in the top three rows of the figure are wrappers for the general macro `mosfet(`*linespec*`,R,`*characters*`,E)`. The third argument of this macro is a subset of the characters {`BDEFGLQRSTXZ`}, each letter corresponding to a diagram component as shown in the bottom row of the figure. Preceding the characters `B`, `G`, and `S` by `u` or `d` adds an up or down arrowhead to the pin, and preceding `T` by `d` negates the pin. This system allows considerable freedom in choosing or customizing components, as illustrated in Figure 31.

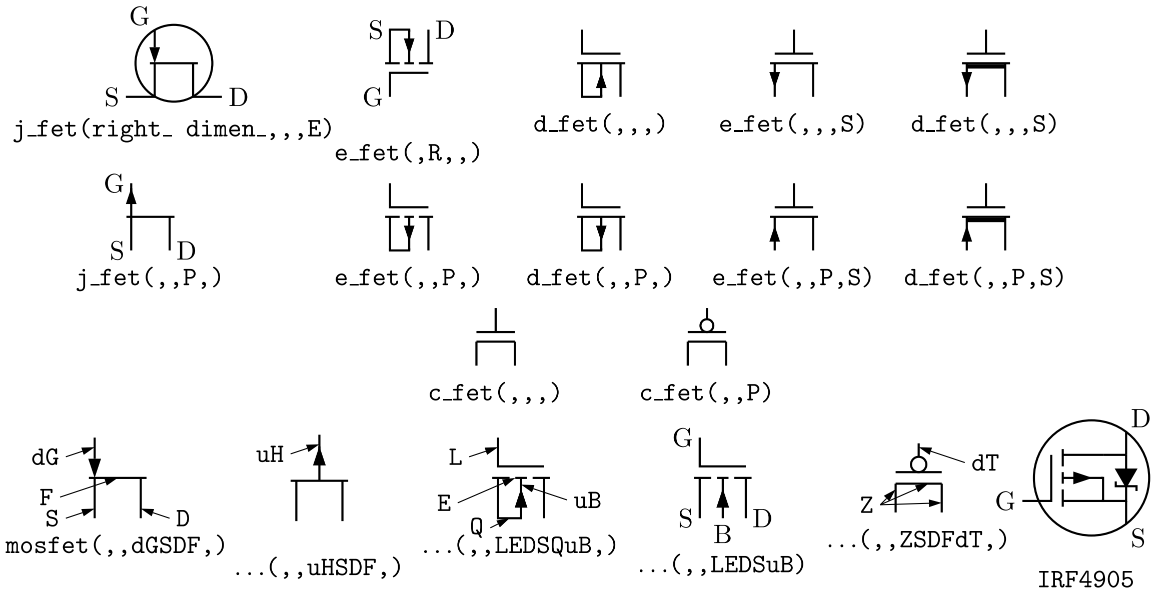

Figure 31: JFET, insulated-gate enhancement and depletion MOSFETS, and simplified versions. These macros are wrappers that invoke the `mosfet` macro as shown in the bottom row. The two lower-right examples show custom devices, the first defined by omitting the substrate connection, and the second defined using a wrapper macro.

The number of possible semiconductor symbols is very large, so these macros must be regarded as prototypes. Often an element is a minor modification of existing elements. For example, the `thyristor(`*linespec*`, `*chars*`)` macro illustrated in Figure 30 is derived from the diode and bipolar transistor macros. Another example is the `tgate` macro shown in Figure 32, which also shows a pass transistor.

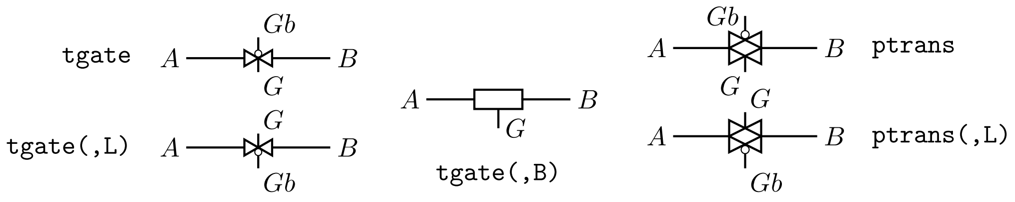

Figure 32: The `tgate(`*linespec*`, [B][R|L])` element, derived from a customized diode and `ebox`, and the `ptrans(`*linespec*`, [R|L])` macro. These are not two-terminal elements, so the *linespec* argument defines the direction and length of the line from A to B but not the element position.

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 33. This figure also illustrates now elements and labels can be colored using the macro

`rgbdraw(`*r*`, `*g*`, `*b*`, `*drawing commands*`)`

where the *r, g, b* values are in the range 0 to 1 to specify the rgb color. This macro is a wrapper for the following, which may be more convenient if many elements are to be given the same color:

`setrgb(`*r*`, `*g*`, `*b*`)`
drawing commands
`resetrgb`

A macro is also provided for colored fills:

`rgbfill(`*r*`, `*g*`, `*b*`, `*drawing commands*`)`

These macros depend heavily on the postprocessor and are intended only for PSTricks, Ti*k*z PGF, MetaPost, and the Postscript output of dpic.

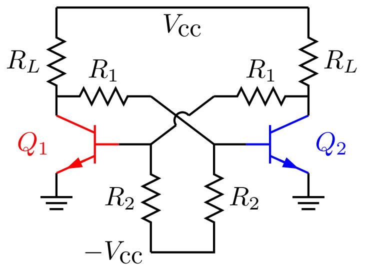

Figure 33: Bipolar transistor circuit, illustrating `crossover` and colored elements.

6 Directions, rotations, and macro-level looping

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 respectively as the macros `up_, down_, left_, right_`.

To draw circuit objects in other than the standard four directions, the macros `Point_(`*degrees*`)`, `point_(`*radians*`)`, and `rpoint_(`*rel linespec*`)` re-define the entries `m4a_`, `m4b_`, `m4c_`, `m4d_` of a transformation matrix, which is used for rotations and, potentially, for more general transformations. The macro `eleminit_` in the two-terminal elements invokes `rpoint_` with a specified or default *linespec* to establish element length and direction. As shown in Figure 34, '`Point_(-30); resistor`' draws a resistor along a line with slope of -30 degrees, and '`rpoint_(to Z)`' sets the current direction cosines to point from the current location to location Z. Macro `vec_(x,y)` evaluates to the position `(x,y)` rotated as defined by the argument of the previous `Point_`, `point_` or `rpoint_` command.

```
% ‘Oblique.m4’
.PS
cct_init

Ct:dot; Point_(-60); capacitor(,C); dlabel(0.12,0.12,,,C_3)
Cr:dot; left_; capacitor(,C); dlabel(0.12,0.12,C_2,,)
Cl:dot; down_; capacitor(from Ct to Cl,C); dlabel(0.12,-0.12,,,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-(R.x-Cr.x,Cr.y-R.y) )

   inductor(from L to Cl); dlabel(0,-0.12,,L_2,)
   right_; resistor(from L to R); rlabel(,R_2,)
   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
```

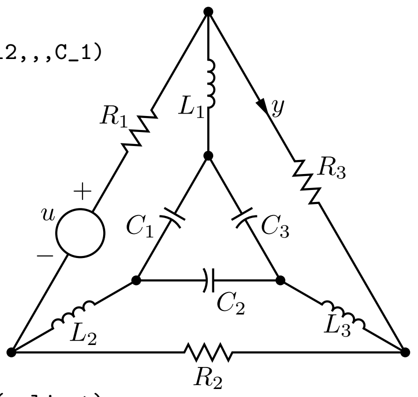

Figure 34: Illustrating elements drawn at oblique angles.

The principal device used to define relative locations in the circuit macros is `rvec_(x,y)`, which evaluates to position `Here + vec_(x,y)`. Thus, `line to rvec_(x,0)` draws a line of length `x` in the current direction.

Figure 34 illustrates that some hand-placement of labels using `dlabel` may be useful when elements are drawn obliquely. The figure also illustrates that any commas within m4 arguments must be treated specially because the arguments are separated by commas. Argument commas are protected either by parentheses as in `inductor(from Cr to Cr+vec_(elen_,0))`, or by multiple single quotes as in `‘‘,’’`, as necessary. Commas also may be avoided by writing `0.5 between L and T` instead of `0.5<L,T>`.

Sequential location names such as *In1*, *In2*, ... in logic and other diagrams can be generated automatically at the m4 processing stage. The `libgen` library defines the macro

`for_(`*start*`,` *end*`,` *increment*`,` `‘`*actions*`’)`

for this and other purposes. Nested loops are allowed and the innermost loop index variable is `m4x`. The first three arguments must be integers and the *end* value must be reached exactly; for example, `for_(1,3,2,‘print In‘’m4x’)` prints locations *In1* and *In3*, but `for_(1,4,2,‘print In‘’m4x’)` does not terminate since the index takes on values 1, 3, 5, ...

Repetetive actions can also be performed with the the `libgen` macro

`Loopover_(‘`*variable*`’,` *actions*`,` *value1*`,` *value2*`, ...)`

which evaluates *actions* for each instance of *variable* set to *value1*, *value2*, ...

7 Logic gates

Figure 35 shows the basic logic gates included in library `liblog.m4`. Gate macros have an optional argument, an integer N from 0 to 16, defining locations `In1`, ... `In`N, as illustrated for the NOR gate in the figure. By default $N = 2$, except for macros `NOT_gate` and `BUFFER_gate`, which have one input `In1` unless they are given a first argument, which is treated as the line specification of a two-terminal element.

Negated inputs or outputs are marked by circles drawn by the `NOT_circle` macro. The name marks the point at the outer edge of the circle and the circle itself has the same name prefixed by

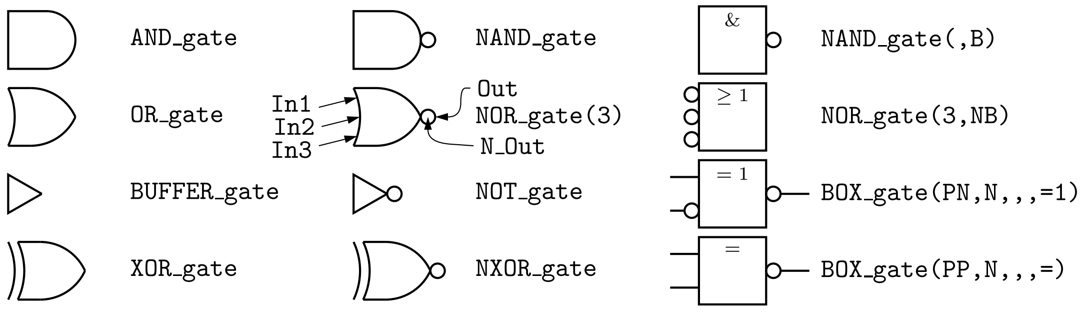

Figure 35: Basic logic gates. The input and output locations of a three-input NOR gate are shown. Inputs are negated by including an N in the second argument letter sequence. A B in the second argument produces a box shape as shown in the rightmost column, where the second example has AND functionality and the bottom two are examples of exclusive OR functions.

N_. For example, the output circle of a nand gate is named N_Out and the outermost point of the circle is named Out. The macro IOdefs creates a sequence of named outputs.

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.

Including an N in the second argument character sequence of any gate negates the inputs, and including B in the second argument invokes the general macro BOX_gate([P|N]...,[P|N],*horiz size*,*vert size*,*label*), which draws box gates. Thus, BOX_gate(PNP,N,,8,\geq 1) creates a gate of default width, eight L_units height, negated output, three inputs with the second negated, and internal label "≥ 1". If the fifth argument begins with sprintf or a double quote then the argument is copied literally; otherwise it is treated as scriptsize mathematics.

Beyond a default number (6) of inputs, the gates are given wings as illustrated in Figure 36.

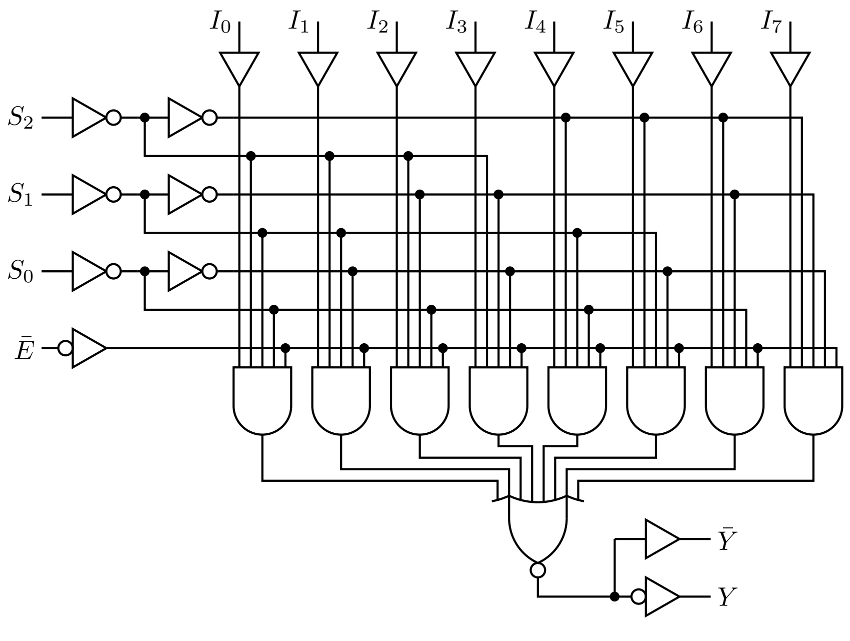

Figure 36: Eight-input binary multiplexer circuit with for_ looping in the source, showing a gate with wings.

Input locations retain their positions relative to the gate body regardless of gate orientation, as shown in Figure 37.

```
% ‘FF.m4’
.PS
log_init
S: NOR_gate
  left_
R: NOR_gate at S+(0,-L_unit*(AND_ht+1))
  line from S.Out right L_unit*3 then down S.Out.y-R.In2.y then to R.In2
  line from R.Out left L_unit*3 then up S.In2.y-R.Out.y 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
```

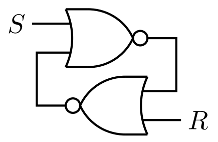

Figure 37: *SR* flip-flop.

Figure 38 shows a multiplexer block with variations, and the macro `FlipFlop(D|T|RS|JK,` *label*, *boxspec*`)`, which is a wrapper for the more specific `FlipFlop6(`*label*, *spec*, *boxspec*`)` and `FlipFlopJK(`*label*, *spec*, *boxspec*`)` macros. Pins on the latter two can be omitted or negated according to their second argument. The second argument of `FlipFlop6`, for example, contains `NQ, Q, CK, S, PR, CLR` to include these pins. Preceding any of these with `n` negates the pin. The substring `lb` is included to write labels on the pins. Any other substring applies to the top left pin, with `.` equating to a blank. Thus, the second argument can be used to customize the flip-flop.

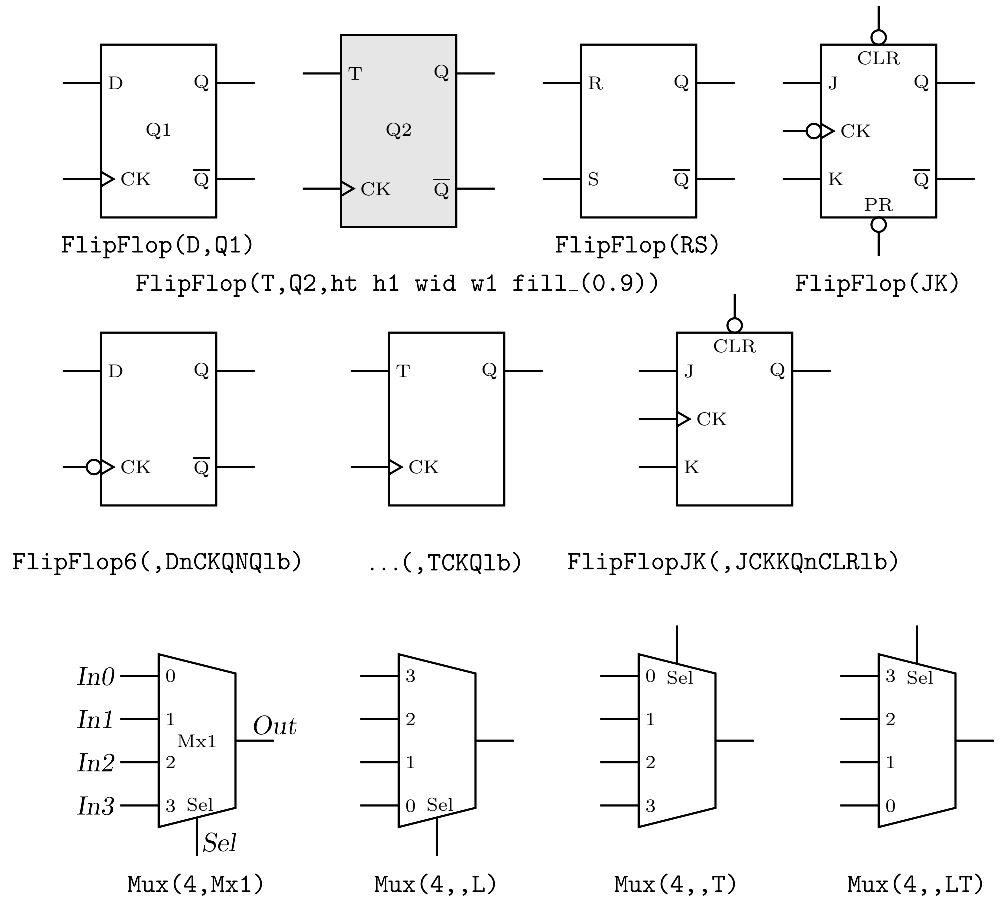

Figure 38: The `FlipFlop` and `Mux` macros, with variations.

Customized gates can be defined simply. For example, the following code defines the custom flip-flops in Figure 39.

```
define(‘customFF’, ‘[ Chip: box wid 10*L_unit ht FF_ht*L_unit
    ifelse(‘$1’,1,‘lg_pin(Chip.se+svec_(0,int(FF_ht/4)),lg_bartxt(Q),PinNQ,e)’)
```

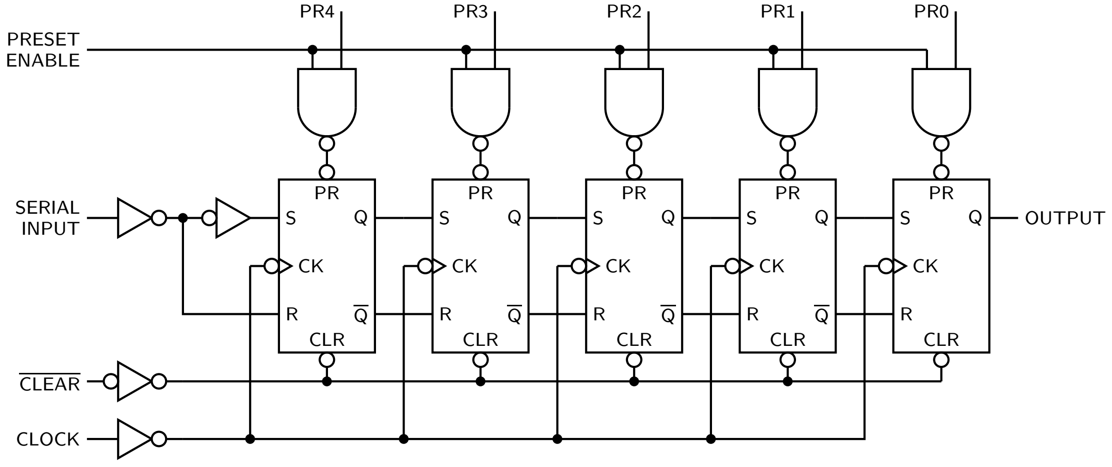

Figure 39: A 5-bit shift register.

```
lg_pin(Chip.ne-svec_(0,int(FF_ht/4)),Q,PinQ,e)
lg_pin(Chip.w,CK,PinCK,wEN)
lg_pin(Chip.n,PR,PinPR,nN)
lg_pin(Chip.s,CLR,PinCLR,sN)
lg_pin(Chip.sw+svec_(0,int(FF_ht/4)),R,PinR,w)
lg_pin(Chip.nw-svec_(0,int(FF_ht/4)),S,PinS,w) ]')
```

This definition makes use of macros `L_unit` and `FF_ht` that predifine dimensions and the logic-pin macro `lg_pin(`*location, printed label, pin name, type*`)`. The pin $\overline{\texttt{Q}}$ is drawn only if the macro argument is 1.

For hybrid applications, the `dac` and `adc` macros are illustrated in Figure 40. The figure shows the default and predifined internal locations, the number of which can be specified as macro arguments.

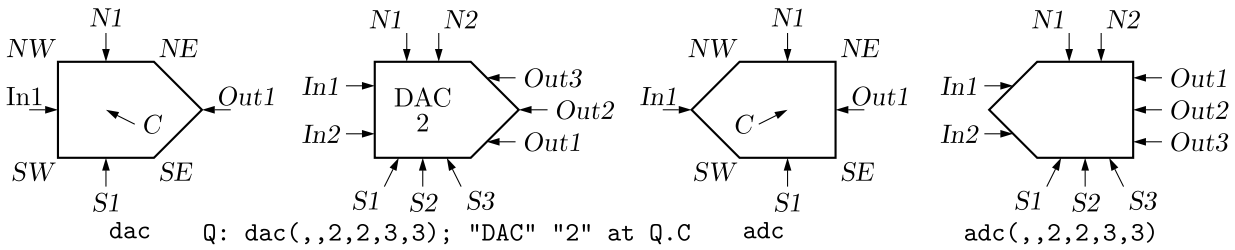

Figure 40: The `dac(`*width,height*`,nIn,nN,nOut,nS)` and `adc(`*width,height*`,nIn,nN,nOut,nS)` macros.

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

In addition to the logic gates described here, some experimental IC chip diagrams are included with 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 scaling of circuit elements will be described first, then the pic scaling facilities.

8.1 Circuit scaling

The circuit elements all have default dimensions that 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.

8.2 Pic scaling

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

1. When generating final output after reading the `.PE` line, pic processors divide distances and sizes by the value of the environmental parameter `scale`, which is 1 by default. Therefore, the effect of assigning a value to `scale` at the beginning of the diagram is to change the drawing unit (initially 1 inch) throughout the figure. For example, 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 are allowed
   ...
   ```

   The default sizes of pic objects are redefined by assigning new values to the environmental parameters `arcrad, arrowht, arrowwid, boxht, boxrad, boxwid, circlerad, dashwid, ellipseht, ellipsewid, lineht, linewid, moveht, movewid, textht,` and `textwid.` The ...`ht` and ...`wid` parameters refer to the default sizes of vertical and horizontal lines, moves, etc., except for `arrowht` and `arrowwid`, which are arrowhead dimensions. The `boxrad` parameter can be used to put rounded corners on boxes. Assigning a new value to `scale` also multiplies all of these parameters except `arrowht, arrowwid, textht,` and `textwid` by the new value of `scale` (gpic multiplies them all). Therefore, objects drawn to default sizes are unaffected by changing `scale` at the beginning of the diagram. To change default sizes, redefine the appropriate parameters explicitly.

2. 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. Line thickness, text size, and dpic arrowheads are unaffected by this scaling.

   If the final picture width exceeds `maxpswid`, which has a default value of 8.5, then the picture is scaled to this size. Similarly, if the height exceeds `maxpsht` (default 11), then the picture is scaled to fit. These parameters can be assigned new values as necessary, for example, to accommodate landscape figures.

3. The finished size of typeset text is independent of pic variables, but can be determined as in Section 10. Then, `"text" wid` x `ht` y tells pic the size of `text`, once the printed width x and height y have been found.

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

9 Writing macros

The m4 language is quite simple and is described in numerous documents such as the original reference [8] 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.

It may not be necessary to define your own macro if all that is needed is a small addition to an existing element that is defined in an enclosing `[ ]` block. After the element arguments are expanded, one argument beyond the normal list is automatically expanded before exiting the block. This extra argument can be used to embellish the element. The process is mentioned with respect to opamps on page 14 and nports on page 15.

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. A few examples will be given.

Example 1: Custom two-terminal elements can often be defined by writing a wrapper for an existing element. For example, an enclosed thermal switch can be defined as shown in Figure 41.

```
define(‘thermalsw’,
 ‘dswitch(‘$1’,‘$2’,WDdBT)
  circle rad distance(M4T,last line.c) at last line.c’)
```

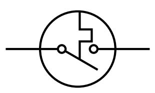

Figure 41: A custom thermal switch defined from the `dswitch` macro.

Example 2: In the following, 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 42.

```
% ‘Tline.m4’
.PS
cct_init
hgt = elen_*1.5
ewd = dimen_*0.9

define(‘sresistor’,‘resistor(right_ ewd); llabel(,r)’)
define(`sinductor',`inductor(right_ ewd,W); llabel(,L)')
define(`tsection',`sinductor
  { dot; line down_ hgt*0.25; dot
    gpar_( resistor(down_ hgt*0.5); rlabel(,R),
           capacitor(down_ hgt*0.5); rlabel(,C))
    dot; line down_ hgt*0.25; 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
```

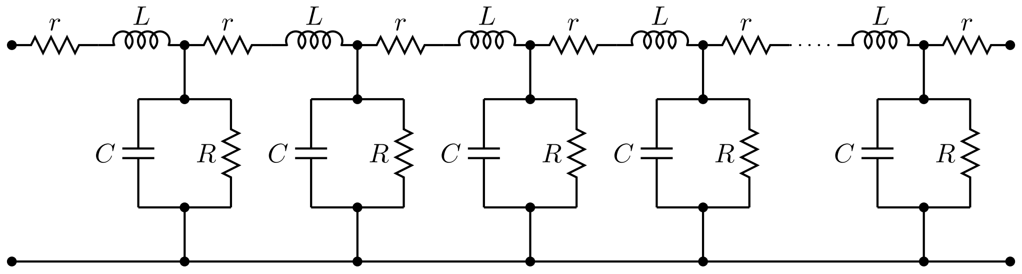

Figure 42: A lumped model of a transmission line, illustrating the use of custom macros.

Example 3: Repeated subcircuits might have different orientations. Suppose that a simple opamp subcircuit might have to be drawn in any direction and possibly oriented to the right with respect to the drawing direction instead of the left default. The subcircuit will be placed in a [ ] block, with internal points *In*, *Out*, and *G*. The macro interface could be something like the following:

`fbfilter( U|D|L|R|`*degrees*`, [L|R], `*opamp label*`, `*C label*`, `*R label* `)`

The first argument specifies the drawing direction as for the `antenna` macro, for example. Setting the second argument to `R` specifies right orientation, and the last three arguments are labels for three internal elements. Two instances of this subcircuit are drawn and placed by the following code, with the result shown in Figure 43.

```
F1: fbfilter(,,K_3,C_{24},R_4)
  ground(at F1.G)
  dot(at F1.In); line up_ elen_/4
F2: fbfilter(L,R,K_2,C_{23},R_3) with .In at F1.In
  ground(at F2.G)
```

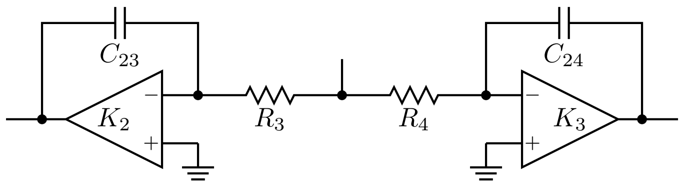

Figure 43: Showing the result of two invokations of the `fbfilter` macro, with labels.

A draft macro for the subcircuit follows:

```
define(`fbfilter',
`[ direction_(ifelse(`$1',,0,`$1')) # Process arg 1, default to the right
   eleminit_                        # Assign rp_ang, rp_len
   tmpang = rp_ang                  # Save rp_ang
   hunit = elen_                    # Dimension parameters
   vunit = ifinstr(`$2',R,-)elen_/2
 K: opamp(,,,,`$2')
   move to K.In`'ifinstr(`$2',R,2,1); line to rvec_(-hunit/4,0)
 J: dot
 R: resistor(to rvec_(-elen_,0)); point_(tmpang) # Reset rp_ang
 In: Here
   move to K.In`'ifinstr(`$2',R,1,2); line to rvec_(-hunit/4,0)
 G: Here
   dot(at K.Out)
   { line to rvec_(hunit/4,0)
 Out: Here }
   line to rvec_(0,vunit)
 C: capacitor(to rvec_(-distance(K.Out,0.5 between J and G),0)); point_(tmpang)
   line to J
   ifelse(`$3',,,"$`$3'$" at K.C)   # Add the labels if non-blank.
   ifelse(`$4',,,"$`$4'$" at C+vec_(0,-vunit/3))
   ifelse(`$5',,,"$`$5'$" at R+vec_(0,-vunit/4))
   ]')
```

The drawing direction is unknown when the macro is defined, so the macros `vec_` and `rvec_` are used for drawing lines and elements. Thus, (`vec_`(*x*,*y*)) is position (*x*,*y*) rotated by angle `rp_ang`. A side effect of drawing a two-terminal element is to change the drawing direction (in conformity with the pic language), so the angle must be saved and reset as needed. Normally, the block will be placed by specifying the position of one of its defined points; by default it will be placed as if it were a box.

10 Interaction with LaTeX

The sizes of typeset labels and other TeX boxes are generally unknown prior to processing the diagram by LaTeX. Although they are not needed for many circuit diagrams, these sizes may be required explicitly for calculations or implicitly for determining the diagram bounding box. For example, the text sizes in the following example affect the total size of the diagram:

```
.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 diagram is generated using default text dimensions. One solution is to measure the text sizes by hand and include them literally, thus:

```
  "Left text" wid 38.47pt__ ht 7pt__ at B.w rjust
```

but this is tedious.

A better solution to this problem is to process the diagram twice. The diagram source is processed as usual by m4 and a pic processor, and the main document source is LaTeXed to input the diagram and format the text, and also to write the required dimensions into a supplementary file. Then the diagram source is processed again, reading the required dimensions from the supplementary file and producing a diagram ready for final LaTeXing. This hackery is summarized below, with an example in Figure 44.

- Put `\usepackage{boxdims}` into the document source.

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

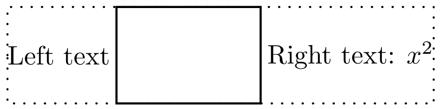

Figure 44: The macro `s_box` sets string dimensions automatically when processed twice. If two or more arguments are given to `s_box`, they are passed through `sprintf`. The dots show the figure bounding box.

- Insert the following at the beginning of the diagram source, where *jobname* is the name of the main LaTeX file:
  `sinclude(`*jobname*`.dim)`
  `s_init(`*unique name*`)`
- Use the macro `s_box(`*text*`)` to produce typeset text of known size as shown in Figure 44; alternatively, invoke the macros `\boxdims` and `boxdim` described later.

The macro `s_box(`*text*`)` evaluates initially to
   `"\boxdims{`*name*`}{`*text*`}" wid boxdim(`*name*`,w) ht boxdim(`*name*`,v)`
On the second pass, this is equivalent to
   `"`*text*`" wid` *x* `ht` *y*
where *x* and *y* are the typeset dimensions of the LaTeX input text. If `s_box` is given two or more arguments as in Figure 44 then they are processed by `sprintf`.

The argument of `s_init`, which should be unique within *jobname*`.dim`, is used to generate a unique `\boxdims` first argument for each invocation of `s_box` in the current file. 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 of `s_box` is LaTeX source, make a rule of quoting it to avoid comma and name-clash problems. For convenience, the macros `s_ht`, `s_wd`, and `s_dp` evaluate to the dimensions of the most recent `s_box` string or to the dimensions of their argument names, if present.

The file `boxdims.sty` distributed with this package should be installed where LaTeX can find it. The essential idea is to define a two-argument LaTeX macro `\boxdims` that 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). These definitions are required by the `boxdim` macro described below.

The LaTeX macro
   `\boxdimfile{`*dimension file*`}`
is used to specify an alternative to *jobname*`.dim` as the dimension file to be written. This simplifies cases where *jobname* is not known in advance or where an absolute path name is required.

Another simplification is available. Instead of the `sinclude(`*dimension file*`)` line above, the dimension file can be read by m4 before reprocessing the source for the second time:
   `m4` *library files dimension file diagram source file* ...

Objects can be tailored to their attached text by invoking `\boxdims` and `boxdim` explicitly. The small source file in Figure 45, for example, produces the box in the figure.

The figure is processed twice, as described previously. The line `sinclude(`*jobname*`.dim)` reads the named file if it exists. The macro `boxdim(`*name,suffix,default*`)` from `libgen.m4` expands the expression `boxdim(Q,w)` to the value of `Q_w` if it is defined, else to its third argument if defined, else

```
% ‘eboxdims.m4’
.PS
sinclude(CMman.dim)  # The main input file is CMman.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
```

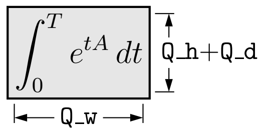

Figure 45: Fitting a box to typeset text.

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.

Sometimes a label needs a plain background in order to blank out previously drawn components overlapped by the label, as shown on the left of Figure 46. The technique illustrated in Figure 45

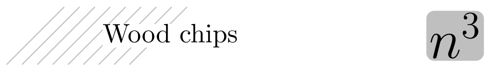

Figure 46: Illustrating the `f_box` macro.

is automated by the macro `f_box(`*boxspecs*, *label arguments*`)`. For the special case of only one argument, e.g., `f_box(Wood chips)`, this macro simply overwrites the label on a white box of identical size. Otherwise, the first argument specifes the box characteristics (except for size), and the macro evaluates to

`box` *boxspecs* `s_box(`*label arguments*`)`.

For example, the result of the following command is shown on the right of Figure 46.

`f_box(color "lightgray" thickness 2 rad 2pt__,"\huge$n^{%g}$",4-1)`

More tricks can be played. The example

`Picture: s_box(‘\includegraphics{`*file*`.eps}’) with .sw at` *location*

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

11 PSTricks and other tricks

This section applies only to a pic processor (dpic) that is capable of producing output compatible with PSTricks, Ti*k*z PGF, or, in principle, other graphics postprocessors.

By using `command` lines, or simply by inserting LaTeX graphics directives along with strings to be formatted, one can mix arbitrary PSTricks (or other) commands with m4 input to create complicated effects.

Some commonly required effects are particularly simple. For example, the rotation of text by PSTricks postprocessing 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 contains both horizontal text and text rotated 90° along the vertical line. This rotation of text is also implemented by the macro `rs_box`, which is similar to `s_box` but rotates its text argument by 90°, a default angle that can be changed by preceding invocation with `define(‘text_ang’,`*degrees*`)`. The `rs_box` macro requires either PSTricks or Ti*k*z PGF and, like `s_box`, it calculates the size of the resulting text box but requires the diagram to be processed twice.

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 "‘}%’"
```

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*`)`.

The macros `shade(`*gray value,closed line specs*`)` and `rgbfill(`*red value, green value, blue value, closed line specs*`)` can be invoked to accomplish the same effect as the above fill example, but are not confined to use only with PSTricks.

Since arbitrary LaTeX can be output, either in ordinary strings or by use of `command` output, complex examples such as found in reference [4], for example, can be included. The complications are twofold: LaTeX and dpic may not know the dimensions of the formatted result, and the code is generally unique to the postprocessor. Where postprocessors are capable of equivalent results, then macros such as `rs_box`, `shade`, and `rgbfill` mentioned previously can be used to hide code differences.

12 Web documents, `pdf`, and alternative output formats

Circuit diagrams contain graphics and symbols, and the issues related to web publishing are similar to those for other mathematical documents. Here the important factor is that gpic `-t` generates output containing tpic `\special` commands, which must be converted to the desired output, whereas dpic can generate several alternative formats, as shown in Figure 47. One of the easiest methods for producing web documents is to generate postscript as usual and to convert the result to pdf format with Adobe Distiller® or equivalent.

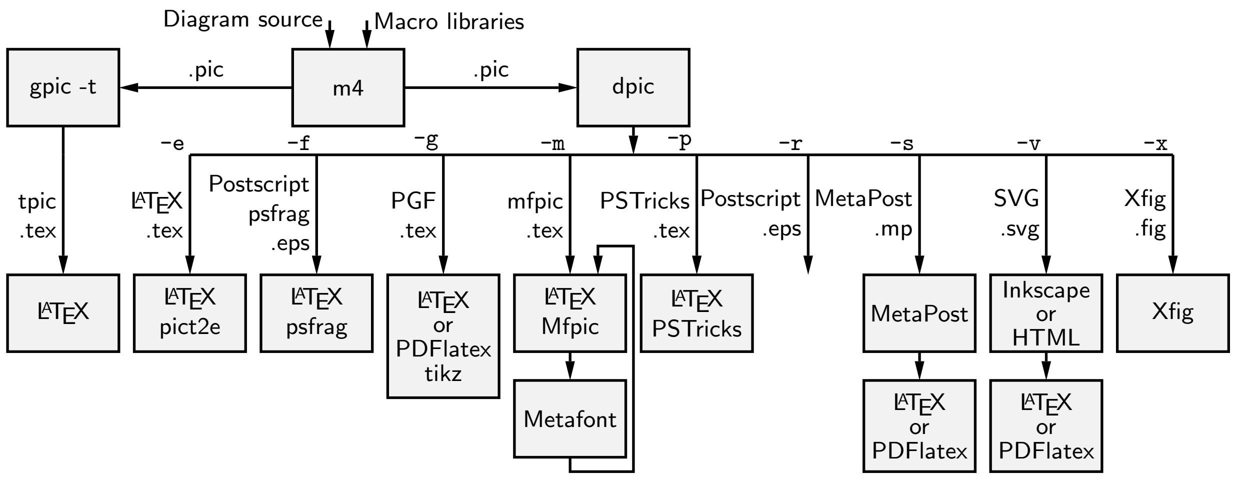

Figure 47: Output formats produced by gpic `-t` and dpic. SVG output can be read by Inkscape or used directly in web documents.

PDFlatex produces pdf without first creating a postscript file but does not handle tpic `\special`s, so dpic must be installed.

Most PDFLatex distributions are not directly compatible with PSTricks, but the Ti*k*z PGF output of dpic is compatible with both LaTeX and PDFLatex. Several alternative dpic output formats such as mfpic and MetaPost also work well. To test MetaPost, create a file *filename*`.mp` containing appropriate header lines, for example:

```
verbatimtex
\documentclass[11pt]{article}
\usepackage{times,boxdims,graphicx}
\boxdimfile{tmp.dim}
\begin{document} etex
```

Then append one or more diagrams by using the equivalent of

`m4` <*path*>`mpost.m4` *library files* *diagram*`.m4 | dpic -s >>` *filename*`.mp`

The command "`mpost --tex=latex` *filename*`.mp end`" processes this file, formatting the diagram text by creating a temporary `.tex` file, LaTeXing it, and recovering the `.dvi` output to create *filename*`.1` and other files. If the `boxdims` macros are being invoked, this process must be repeated to handle formatted text correctly as described in Section 10. In this case, either put `sinclude(tmp.dim)` in the diagram `.m4` source or read the `.dim` file at the second invocation of m4 as follows:

`m4` <*path*>`mpost.m4` *library files* `tmp.dim` *diagram*`.m4 | dpic -s >>` *filename*`.mp`

On some operating systems, the absolute path name for `tmp.dim` has to be used to ensure that the correct dimension file is written and read. This distribution includes a `Makefile` that simplifies the process; otherwise a script can automate it.

Having produced *filename*`.1`, rename it to *filename*`.mps` and, *voilà,* you can now run PDFlatex on a `.tex` source that includes the diagram using `\includegraphics{`*filename*`.mps}` in the usual way.

The dpic processor is capable of other output formats, as illustrated in Figure 47 and in example files included with the distribution. The LaTeX drawing commands alone or with `eepic` or `pict2e` extensions are suitable only for simple diagrams.

13 Developer's notes

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, drawings I have had to produce since, and suggestions received from other people. The interpreter has been upgraded over time to generate mfpic, MetaPost [6], raw Postscript, Postscriptwith `psfrag` tags, and PSTricks output, the latter my preference because of its quality and flexibility, including facilities for colour and rotations, together with simple font selection. Ti*k*Z PGF output, which combines the simplicity of PSTricks with PDFlatex compatibility has been added. In addition, xfig-compatible output was added early on to allow diagram creation both by pic programming and interactive graphics. Most recently, SVG output has been added, and seems suitable for producing web diagrams directly and for further editing by the Inkscape interactive grapics editor. 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? It would depend on the context, but pic remains a good choice for line drawings since it is easy to learn and read but powerful enough for coding the geometrical calculations required for precise component sizing and placement. However, the main value of this distribution is not in the use of a specific language but in the element data encoded in the macros, which have been developed and refined over nearly two decades. Some of them have become less readable as more options and flexibility have been added, and if starting over today, perhaps I would change some details. No choice of tool is without compromise, and producing good graphics seems to be time-consuming, no matter how it is done.

The dpic interpreter has several output-format options that may be useful. The `eepicemu` and `pict2e` extensions of the primitive LaTeX picture objects are supported. The mfpic output allows the production of Metafont alphabets of circuit elements or other graphics, thereby essentially removing dependence on device drivers, but with the complication of treating every alphabetic component as a TeX box. The xfig output allows elements to be precisely defined with dpic and interactively placed with xfig. Similarly, the SVG output can be read directly by the Inkscape graphics editor, but SVG can also be used directly for web pages. Dpic will also generate low-level MetaPost or Postscript code, so that diagrams defined using pic can be manipulated and combined with others. The Postscript output is compatible with CorelDraw®, and by extension to Adobe Illustrator®. With raw Postscript output the user is responsible for ensuring that the correct fonts are provided and for formatting labels.

Many thanks to the people who continue to send comments, questions, and, occasionally, bug fixes. What began as a tool for my own work grew into a hobby that has persisted, thanks to your help and advice.

14 Bugs

The distributed macros are not written for maximum robustness. Arguments could be entered in a key–value style (for example, `resistor(up_ elen_,style=N;cycles=8)` instead of by positional parameters. 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.

In response to suggestions, some of the macros have been modified to allow easier customization to forms not originally anticipated, but these modifications are not complete.

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

1. **Misconfiguration:** One of the following configuration files *must* be read by m4 before any of the other files, depending on the required form of pic output: `gpic.m4, pstricks.m4, postscript.m4, pgf.m4, mpost.m4, mfpic.m4, svg.m4,` or `xfig.m4.` The package default is to read `gpic.m4.` The processor options must be set correspondly, `gpic -t` for `gpic.m4` and, most often, `dpic -p` or `dpic -g` when dpic is employed. For example, the pipeline for PSTricks output from file `circuit.m4` is

   ```
   m4 <path>pstricks.m4 <path>libcct.m4 circuit.m4 | dpic -p > circuit.tex
   ```

   but for Ti*k*z PGF processing, the configuration file and dpic option have to be changed:

   ```
   m4 <path>pgf.m4 <path>libcct.m4 circuit.m4 | dpic -g > circuit.tex
   ```

   Any non-default configuration file must be processed explicitly as illustrated above. To redefine the default behaviour, change the `include` statements near the top of the libraries. The top-level makefile automates these changes.

2. **Initialization:** If the first element macro evaluated is not 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, make sure that the line

   ```
   cct_init
   ```

   appears immediately after the .PS line or prior to the first block. It is entirely permissible to modify `cct_init` to include customized diagram initializations such as the `thicklines_` statement. For completeness, macros `gen_init, log_init, darrow_init` are also provided for cases where the circuit library is not needed.

3. **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.

4. **Commas:** Macro arguments are separated by commas, so any comma that is 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.

5. **Default directions and lengths:** The *linespec* argument of element macros requires both a direction and a length, and if either is omitted, a default value is used. Writing

   ```
   source(up_)
   ```

   draws a source up a distance equal to the current `lineht` value, which may cause confusion. Writing

   ```
   source(0.5)
   ```

   draws a source of length 0.5 units in the current pic default direction, which is one of `right,` `left,` `up,` or `down.` It is usually better to specify both the direction and length of an element, thus:

   ```
   source(up_ elen_).
   ```

   The effect of a *linespec* argument is independent of any direction set using the `Point_` or similar macros. To draw an element at an obtuse angle (see Section 6) try, for example,

   ```
   Point_(45); source(to rvec_(0.5,0))
   ```

6. **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 only subtlety required in 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 or where the quotes could be omitted.

7. **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$"`.

8. **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}
```

9. **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.

10. **Position of elements that are not 2-terminal:** The *linespec* argument of elements defined in `[ ]` blocks must be understood as defining a direction and length, but not the position of the resulting block. In the pic language, objects inside these brackets are placed by default *as if the block were a box.* Place the element by its compass corners or defined interior points as described in the first paragraph of Section 5 on page 13, for example

    `igbt(up_ elen_) with .E at (1,0)`

11. **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.

12. **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 parameters, 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.

13. **Buffer overflow:** For some m4 implementations, the error message `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.

15 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 sources in the `examples` directory contain additional macros, such as for flowcharts, Boolean logic, and binary trees.

Internal macros defined within the libraries begin with the characters m4 or M4 and, for the most part, are not listed here.

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

`AND_gate(`*n*`,N)`	log	basic 'and' gate, 2 or n inputs; N=negated input
`AND_gen(`*n*`,`*chars*`,[`*wid*`,[`*ht*`]])`	log	general AND gate: n=number of inputs ($0 \leq n \leq 16$); *chars:* B=base and straight sides; A=Arc; [N]NE,[N]SE,[N]I,[N]N,[N]S=inputs or circles; [N]O=output; C=center

`AND_ht`	log	height of basic 'and' and 'or' gates in `L_units`
`AND_wd`	log	width of basic 'and' and 'or' gates in `L_units`
`BOX_gate(`*inputs,output,swid,sht,label*`)`	log	output=[`P`|`N`], inputs=[`P`|`N`]..., sizes swid and sht in `L_units`
`BUFFER_gate(`*linespec*`, N)`	log	basic buffer, 1 input or as a 2-terminal element, N=negated input
`BUFFER_gen(`*chars,wd,ht*`,[N|P]*,[N|P]*,[N|P]*)`	log	general buffer, *chars:* `T`=triangle, `[N]O`=output location `Out` (`NO` draws circle `N_Out`); `[N]I, [N]N, [N]S, [N]NE, [N]SE` input locations; `C`=centre location. Args 4-6 allow alternative definitions of respective `In, NE,` and `SE` argument sequences
`BUF_ht`	log	basic buffer gate height in `L_units`
`BUF_wd`	log	basic buffer gate width in `L_units`
`Cos(`*integer*`)`	gen	cosine function, *integer* degrees
`Cosine(` *amplitude*`,` *freq*`,` *time*`,` *phase* `)`	gen	function $a \times \cos(\omega t + \phi)$
`Darlington(`*linespec*`,L|R,P)`	cct	left or right, N- or P-type bipolar Darlington pair, internal locations E, B, C
`E_`	gen	the constant *e*
`Equidist3(`*Pos1, Pos2, Pos3, Result*`)`	gen	Calculates location named *Result* equidistant from the first three positions, i.e. the centre of the circle passing through the three positions
`FF_ht`	cct	flipflop height parameter in `L_units`
`FF_wid`	cct	flipflop width parameter in `L_units`
`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,` *label*`,` *boxspec*`)`	log	flip-flops, *boxspec*=e.g. ht x wid y
`FlipFlop6(`*label*`,` *spec*`,` *boxspec*`)`	log	6-input flip-flops, *spec*=`[[n]NQ][[n]Q][[n]CK][[n]PR][lb] [[n]CLR][[n]S][[n].|D|T|R]` to include and negate pins, `lb` to print labels
`FlipFlopJK(`*label*`,` *spec*`,`*boxspec*`)`	log	JK flip-flop, spec similar to above
`G_hht`	log	gate half-height in `L_units`
`HOMELIB_`	all	directory containing libraries
`H_ht`	log	hysteresis symbol dimension in `L_units`
`Int_`	gen	corrected (old) gpic *int*() function
`IOdefs(`*linespec*`,`*label*`,[P|N]*,L|R)`	log	Define locations *label*`1`, ... *label*`n` along the line; `P`= label only; `N`=with `NOT_circle`; `R`=circle to right of current direction
`Intersect_(`*Name1*`,`*Name2*`)`	gen	intersection of two named lines
`LH_symbol(U|D|L|R|degrees)`	log	logic-gate hysteresis symbol
`Loopover_(`'*variable*'`,`*actions*`,`*value1, value2, ...*`)`		

	gen	Repeat *actions* with *variable* set successively to *value1*, *value2*, . . .
LT_symbol(U|D|L|R|degrees)	log	logic-gate triangle symbol
L_unit	log	logic-element grid size
Max(*arg, arg, . . .*)	gen	Max of an arbitrary number of inputs
Min(*arg, arg, . . .*)	gen	Min of an arbitrary number of inputs
Mux(*n, label*, [L][T])	gen	binary multiplexer, *n* inputs, L reverses pin numbers, T puts Sel pin to top
Mux_ht	cct	Mux height parameter in L_units
Mux_wid	cct	Mux width parameter in L_units
Mx_pins	log	max number of gate inputs without wings
NAND_gate(*n*,N)	log	'nand' gate, 2 or *n* inputs; N=negated input
NOR_gate(*n*,N)	log	'nor' gate, 2 or *n* inputs; N=negated input
NOT_circle	log	'not' circle
NOT_gate(*linespec*,N)	log	'not' gate, 1 input or as a 2-terminal element, N=negated input
NOT_rad	log	'not' radius in absolute units
NXOR_gate(*n*,N)	log	'nxor' gate, 2 or *n* inputs; N=negated input
N_diam	log	diameter of 'not' circles in L_units
N_rad	log	radius of 'not' circles in L_units
OR_gate(*n*,N)	log	'or' gate, 2 or *n* inputs; N=negated input
OR_gen(*n*,*chars*,[*wid*,[*ht*]])	log	general OR gate: *n*=number of inputs ($0 \leq n \leq 16$); *chars:* B=base and straight sides; A=Arcs; [N]NE,[N]SE,[N]I,[N]N,[N]S=inputs or circles; [N]P=XOR arc; [N]O=output; C=center
OR_rad	log	radius of OR input face in L_units
Point_(*integer*)	gen	sets direction cosines in degrees
Rect_(*radius*,*angle*)	gen	(deg) polar-to-rectangular conversion
Sin(*integer*)	gen	sine function, *integer* degrees
View3D	3D	The view vector (triple) defined by setview(*azim*, *elev*). The project macro projects onto the plane perpendicular to this vector
Vperp(*position name*, *position name*)	gen	unit-vector pair CCW-perpendicular to line joining two named positions
XOR_gate(*n*,N)	log	'xor' gate, 2 or *n* inputs; N=negated input
XOR_off	log	XOR and NXOR offset of input face
above_	gen	string position above relative to current direction
abs_(*number*)	gen	absolute value function
adc(*width*,*height*,nIn,nN,nOut,nS)	cct	ADC with defined width, height, and number of inputs In*i*, top terminals N*i*, ouputs Out*i*, and bottom terminals S*i*
amp(*linespec*,*size*)	cct	amplifier
along_(*linear object name*)	gen	short for between name.start and name.end
antenna(at *location*, T, A|L|T|S|D|P|F, U|D|L|R|*degrees*)		

	cct	antenna, without stem for nonblank 2nd arg; A=aerial, L=loop, T=triangle, S=diamond, D=dipole, P=phased, F=fork; up, down, left, right, or angle from horizontal (default -90)
arca(*chord linespec*, ccw|cw, *radius*, *modifiers*)	gen	arc with acute angle (obtuse if radius is negative)
arcd(*center*, *radius*,*start degrees*,*end degrees*)	gen	arc definition (see arcr), angles in degrees
arcr(*center*,*radius*,*start angle*,*end angle*)	gen	arc definition, e.g., arcr(A,r,0,pi_/2) cw ->
arcto(*position 1*,*position 2*,*radius*,[dashed|dotted])	gen	line toward position 1 with rounded corner toward position 2
arrowline(*linespec*)	cct	line (dotted, dashed permissible) with centred arrowhead
b_	gen	blue color value
b_current(*label*,*pos*,In|Out,Start|End,*frac*)	cct	labelled branch-current arrow to *frac* between branch end and body
battery(*linespec*,n,R)	cct	n-cell battery: default 1 cell, R=reversed polarity
beginshade(*gray value*)	gen	begin gray shading, see shade e.g., beginshade(.5); *closed line specs*; endshade
bell(U|D|L|R|*degrees*, *size*)	cct	bell, *In1* to *In3* defined
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
bi_trans(*linespec*,L|R,*chars*,E)	cct	bipolar transistor, core left or right; chars: BU=bulk line, B=base line and label, S=Schottky base hooks, uEn|dEn=emitters E0 to En, uE|dE=single emitter, Cn|uCn|dCn=collectors C0 to Cn; u or d add an arrow, C=single collector; u or d add an arrow, G=gate line and location, H=gate line; L=L-gate line and location, [d]D=named parallel diode, d=dotted connection, [u]T=thyristor trigger line; arg 4 = E: 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
bp__	gen	big-point-size factor, in scaled inches, (*scale/72)
bswitch(*linespec*, [L|R],chars)	cct	pushbutton switch R=right orientation (default L=left); chars: O= normally open, C=normally closed
buzzer(U|D|L|R|degrees, size,[C])	cct	buzzer, *In1* to *In3* defined, C=curved
c_fet(*linespec*,L|R,P)	cct	left or right, plain or negated pin simplified MOSFET
capacitor(*linespec*,*char*[+],R)	cct	capacitor, *char*: F or none=flat plate, C=curved-plate, E=polarized boxed plates, K=filled boxed plates, P=alternate polarized; + adds a polarity sign; arg3: R=reversed polarity
cbreaker(*linespec*, L|R, D)	cct	circuit breaker to left or right, D=dotted

cct_init	cct	initialize circuit-diagram environment
centerline_(*linespec, thickness*|*color, minimum long dash len, short dash len, gap len*	gen	Technical drawing centerline
cintersect(*Pos1, Pos2, rad1, rad2,* [R])	gen	Upper (lower if arg5=R) intersection of circles at *Pos1* and *Pos2*, radius *rad1* and *rad2*
clabel(*label*,*label*,*label*)	cct	centre triple label
consource(*linespec*,V|I|v|i,R)	cct	voltage or current controlled source with alternate forms; R=reversed polarity
corner(*line thickness,color*)	gen	filled square to make square corner at line intersection
contact(O|C,R)	cct	single-pole contact: default double pole or normally open or closed, R=oriented right (default left)
contline(*line*)	gen	evaluates to continue if processor is **dpic**, otherwise to first arg (default line)
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
dac(*width,height*,nIn,nN,nOut,nS)	cct	DAC with defined width, height, and number of inputs In*i*, top terminals N*i*, ouputs Out*i*, and bottom terminals S*i*
d_fet(*linespec*,L|R,P,S,E|S)	cct	left or right, N or P depletion MOSFET, normal or simplified, without or with envelope or thick channel
dabove(at *location*)	darrow	above (displaced dlinewid/2)
darrow(*linespec*, t,t,*width*,*arrowhd wd*,*arrowhd ht*, <- *or* <-| *or* |)	darrow	double arrow, truncated at beginning or end, specified sizes, reversed arrowhead or closed stem
darrow_init	darrow	initialize darrow drawing parameters
dashline(*linespec*,*thickness*|*color*|<->, *dash len, gap len*,G)	gen	dashed line with dash at end (G ends with gap)
dbelow(at *location*)	darrow	below (displaced dlinewid/2)
dcosine3D(i,x,y,z)	3D	extract i-th entry of triple x,y,z
def_bisect	gen	defines the pic procedure bisect (func, xmin, xmax, eps, result) that finds a root of func(arg,value) to precision eps in the interval (xmin,xmax) by the method of bisection
delay(*linespec*,*size*)	cct	delay element
delay_rad_	cct	delay radius
deleminit_	darrow	sets drawing direction for dlines
dend(at *location*)	darrow	close (or start) double line
diff3D(x1,y1,z1,x2,y2,z2)	3D	difference of two triples
diff_(*a*,*b*)	gen	difference function
dimen_	cct	size parameter for circuit elements

`dimension_(`*linespec*`,`*offset*`,`*label*`, D|H|W|`*blank width*`,`*tic offset*`,`*arrowhead* `)`	gen	macro for dimensioning diagrams; *arrowhead*=`-> | <-`
`diode(`*linespec*`,B|CR|D|K|L|LE[R]|P[R]|S|T|V|v|Z,[R][E])`	cct	diode: `B`=bi-directional, `CR`=current regulator, `D`=diac, `K`=open form, `L`=open form with centre line, `LED[R]`=LED [right], `P[R]`=photodiode [right], `S`=Schottky, `T`=tunnel, `V`=varicap, `v`=varicap (curved plate), `Z`=zener; arg 4: `R`=reversed polarity, `E`=enclosure
`dir_`	darrow	used for temporary storage of direction by darrow macros
`distance(`*Position 1*`, `*Position2*`)`	gen	distance between named positions
`direction_(U|D|L|R|`*degrees*`, `*default*`)`	gen	sets current direction up, down, left, right, or angle in degrees
`distance(`*position*`, `*position*`)`	gen	distance between positions
`dlabel(`*long*`,`*lat*`,`*label*`,`*label*`,`*label*`)`	cct	general triple label
`dleft`	darrow	double line left turn
`dline(`*linespec*`,t,t,`*width*`, |-|` *or* `-|` *or* `|-)`	darrow	double line, truncated by half width at either end, closed at either or both ends
`dlinewid`	darrow	width of double lines
`dn_`	gen	down with respect to current direction
`dljust(at `*location*`)`	darrow	ljust (displaced dlinewid/2)
`dn_`	gen	sets down relative to current-direction
`dna_`	cct	characters that determine which components are drawn
`dnm_`	cct	similar to dna_
`dot(at `*location*`,`*radius*`,`*fill*`)`	gen	filled circle (third arg= gray value: 0=black, 1=white)
`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
`drjust(at `*location*`)`	darrow	rjust (displaced dlinewid/2)
`dswitch(`*linespec*`,L|R,W[ud]B[K]`*chars*`)`	cct	SPST switch left or right, W=baseline, B=contact blade, dB=contact blade to the right of drawing direction, K=vertical closing contact line, C = external operating mechanism, D = circle at contact and hinge, (dD = hinge only, uD = contact only) E = emergency button, EL = early close (or late open), LE = late close (or early open), F = fused, H = time delay closing, uH = time delay opening, HH = time delay opening and closing, K = vertical closing contact, L = limit, M = maintained (latched), MM = momentary contact on make, MR = momentary contact on release, MMR = momentary contact on make and release, O = hand operation button, P = pushbutton, T = thermal control linkage, Y = pull switch, Z = turn switch

`dtee([L|R])`	darrow	double arrow tee junction with tail to left, right, or (default) back along current direction
`dtor_`	gen	degrees to radians conversion constant
`dturn(`*degrees ccw*`)`	darrow	turn dline arg1 degrees left (ccw)
`e_`	gen	.e relative to current direction
`e_fet(`*linespec*`,L|R,P,S,E|S)`	cct	left or right, N or P enhancement MOSFET, normal or simplified, without or with envelope or thick channel
`earphone( U|D|L|R|degrees, size)`	cct	earphone, *In1* to *In3* defined
`ebox(`*linespec*`,`*length*`,`*ht*`,`*fill value*`)`	cct	two-terminal box element with adjustable dimensions and fill value 0 (black) to 1 (white)
`elchop(`E`,`A`)`	gen	`chop` for ellipses: evaluates to `chop` r where r is the distance from the centre of ellipse E to the intersection of E with a line to location A; e.g., `line from A to E elchop(E,A)`
`eleminit_(`*linespec*`)`	cct	internal line initialization
`elen_`	cct	default element length
`em_arrows([N|I|E][D],`*angle,length*`)`	cct	radiation arrows, N=nonionizing, I=ionizing, E=simple; D=dot
`endshade`	gen	end gray shading, see `beginshade`
`expe`	gen	exponential, base e
`f_box(`*boxspecs*`,`*text*`,`*expr1*`,`···`)`	gen	like `s_box` but the text is overlaid on a box of identical size. If there is only one argument then the default box is invisible and filed white
`fill_(`*number*`)`	gen	fill macro, 0=black, 1=white
`fitcurve(V,n,[e.g.  dotted],m (default 0))`	gen	Draw a spline through V[m],...V[n]: Works only with dpic (and $n - m > 2$): V[m]:*position*; ... V[n]:*position*
`for_(`*start*`,`*end*`,`*increment*`,`*‘actions’*`)`	gen	integer for loop with index variable `m4x`
`fuse(`*linespec, type, wid, ht*`)`	cct	fuse symbol, type= `A|B|C|D|S|HB|HC` or `dA=D`
`g_`	gen	green color value
`gap(`*linespec*`,`*fill*`,A)`	cct	gap with (filled) dots, A=chopped arrow between dots
`gen_init`	gen	initialize environment for general diagrams (customizable)
`glabel_`	cct	internal general labeller
`gpar_(`*element*`,`*element*`,`*separation*`)`	cct	two same-direction elements in parallel
`gpolyline_(`*fraction*`,`*location*`, ...)`	gen	internal to `gshade`
`grid_(`x`,`y`)`	log	absolute grid location
`ground(at` *location*`, T, N|F|S|L|P|E, U|D|L|R|`*degrees*`)`	cct	ground, without stem for nonblank 2nd arg; `N`=normal, `F`=frame, `S`=signal, `L`=low-noise, `P`=protective, `E`=European; up, down, left, right, or angle from horizontal (default -90)
`gyrator(`*box specs,space ratio,pin lgth,*`[N][V])`		

	cct	Gyrator two-port wrapper for `nport`, `N` omits pin dots; `V` gives a vertical orientation.
`gshade(`*gray value*`,A,B,...,Z,A,B)`	gen	(Note last two arguments). Shade a polygon with named vertices, attempting to avoid sharp corners
`hoprad_`	cct	hop radius in crossover macro
`ht_`	gen	height relative to current direction
`ifdpic(`*if true*`,`*if false*`)`	gen	test if dpic has been specified as pic processor
`ifgpic(`*if true*`,`*if false*`)`	gen	test if gpic has been specified as pic processor
`ifinstr(`*string*`,`*string*`,`*if true*`,`*if false*`)`	gen	test if the second argument is a substring of the first argument
`ifmfpic(`*if true*`,`*if false*`)`	gen	test if mfpic has been specified as pic post-processor
`ifmpost(`*if true*`,`*if false*`)`	gen	test if MetaPost has been specified as pic post-processor
`ifpgf(`*if true*`,`*if false*`)`	gen	test if Ti*k*z PGF has been specified as pic post-processor
`ifpostscript(`*if true*`,`*if false*`)`	gen	test if Postscript (`dpic -r`) has been specified as pic output format
`ifpstricks(`*if true*`,`*if false*`)`	gen	test if PSTricks has been specified as post-processor
`ifroff(`*if true*`,`*if false*`)`	gen	test if **troff** or **groff** has been specified as post-processor
`ifxfig(`*if true*`,`*if false*`)`	gen	test if Fig 3.2 (`dpic -x`) has been specified as pic output format
`igbt(`*linespec*`,L|R,[L][[d]D])`	cct	left or right IGBT, L=alternate gate type, D=parallel diode, dD=dotted connections
`in__`	gen	absolute inches
`inductor(`*linespec*`,W|L,`*n*`,M)`	cct	inductor, arg2: narrow (default), W=wide, L=looped; arg3: *n* arcs (default 4); arg4: M=magnetic core
`inner_prod(`*linear obj*`,`*linear obj*`)`	gen	inner product of (x,y) dimensions of two linear objects
`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
`larrow(`*label*`,->|<-,`*dist*`)`	cct	arrow *dist* to left of last-drawn 2-terminal element
`lbox(`*wid*`,` *ht*`,` *type*`)`	gen	box oriented in current direction, type= e.g. dotted
`left_`	gen	left with respect to current direction
`length3D(x,y,z)`	3D	Euclidean length of triple x,y,z
`lg_bartxt`	log	draws an overline over logic-pin text (except for xfig)
`lg_pin(`*location, logical name, pin label*`, n|e|s|w[N|L|M][E],` *pinno, optlen*`)`	log	comprehensive logic pin; `n|e|s|w`=direction, `N`=negated, `L`=active low out, `M`=active low in, `E`=edge trigger
`lg_pintxt`	log	reduced-size text for logic pins
`lg_plen`	log	logic pin length in in `L_units`
`lin_leng(`*line-reference*`)`	gen	calculate the length of a line
`linethick_(`*number*`)`	gen	set line thickness in points
`ljust_`	gen	ljust with respect to current direction
`llabel(`*label*`,`*label*`,`*label*`)`	cct	triple label on left side of the element

loc_(*x*, *y*)	gen	location adjusted for current direction
log10E_	gen	constant $\log_{10}(e)$
log_init	log	initialize environment for logic diagrams (customizable)
loge	gen	logarithm, base *e*
lp_xy	log	coordinates used by lg_pin
lpop(*xcoord*, *ycoord*, *radius*, *fill*, *zero ht*)	gen	for lollipop graphs: filled circle with stem to (xcoord,zeroht)
lswitch(*linespec*, L|R, *chars*)	cct	knife switch R=right orientation (default L=left); *chars*=[O|C][D] O= opening; C=closing; D=dots
lt_	gen	left with respect to current direction
manhattan	gen	sets direction cosines for left, right, up, down
memristor(*linespec, wid, ht*)	cct	memristor element
microphone(U|D|L|R|degrees, size)	cct	microphone, *In1* to *In3* defined
mm__	gen	absolute millimetres
mosfet(*linespec*,L|R,*chars*,E)	cct	MOSFET left or right, included components defined by characters, envelope. arg 3 chars: [u][d]B: center bulk connection pin; D: D pin and lead; E: dashed substrate; F: solid-line substrate; [u][d]G: G pin to substrate at source; [u][d]H: G pin to substrate at center; L: G pin to channel; Pz: parallel zener diode; Q: connect B pin to S pin; R: thick channel; [u][d]S: S pin and lead u: arrow up; d: arrow down; [d]T: G pin to center of channel d: not circle; X: XMOSFET terminal; Z: simplified complementary MOS
m4xpand(*arg*)	gen	Evaluate the argument as a macro
m4lstring(*arg1*,*arg2*)	gen	expand *arg1* if it begins with sprintf or ", otherwise *arg2*
m4_arrow(*linespec*,*ht*,*wid*)	gen	arrow with adjustable head, filled when possible
m4xtract(‘*string1*’,*string2*)	gen	delete *string2* from *string1*, return 1 if present
n_	gen	.n with respect to current direction
ne_	gen	.ne with respect to current direction
neg_	gen	unary negation
norator(*linespec*,*width*,*ht*)	cct	norator two-terminal element
nport(*box spec*;*other commands, nw,nn,ne,ns,space ratio,pin lgth,style, other commands*)	cct	nport macro (default 2-port)
nterm(*box spec*;*other commands, nw,nn,ne,ns,pin lgth,style, other commands*)	cct	n-terminal box macro (default three pins)
nullator(*linespec*,*width*,*ht*)	cct	nullator two-terminal element
nw_	gen	.nw with respect to current direction
opamp(*linespec*,*label*,*label*,*size*,*chars*, other commands)	cct	operational amplifier with −, + or other internal labels, specified size. *chars:* P= add power connections, R= swap In1, In2 labels, T= truncated point. The first and last arguments allow added customizations
open_arrow(*linespec*,*ht*,*wid*)	gen	arrow with adjustable open head

`par_(`*element*`,`*element*`,`*separation*`)`	cct	two same-direction, same-length elements in parallel
`pc__`	gen	absolute points
`pi_`	gen	π
`pmod(`*integer*`, `*integer*`)`	gen	+ve mod(M,N) e.g., `pmod`$(-3,5)=2$
`point_(`*angle*`)`	gen	(radians) set direction cosines
`polar_(`*x*`,`*y*`)`	gen	rectangular-to polar conversion
`potentiometer(`*linespec*`,`*cycles*`,`*fractional pos*`,`*length*`,`$\cdots$`)`	cct	resistor with taps T1, T2, ... with specified fractional positions and lengths (possibly neg)
`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 onto the plane perpendicular to the view vector with angles defined by `setview(`*azim, elev*`)`
`psset_(`*PSTricks settings*`)`	gen	set PSTricks parameters
`ptrans(`*linespec*`, [R|L])`	cct	pass transistor; `L=` left orientation
`pt__`	gen	TEX point-size factor, in scaled inches, (`*scale/72.27`)
`r_`	gen	red color value
`rarrow(`*label*`,->|<-,`*dist*`)`	cct	arrow *dist* to right of last-drawn 2-terminal element
`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, `R`=oriented to right (default left) of current direction
`resetrgb`	gen	cancel `r_, g_, b_` color definitions
`resistor(`*linespec*`,n|E,`*chars*`)`	cct	resistor, n cycles (default 3), *chars:* `E`=`ebox`, `Q`=offset, `H`=squared, `R`=right-oriented
`reversed(‘`*macro name*`’,args)`	cct	reverse polarity of 2-terminal element
`rgbdraw(`*color triple*`, `*drawing commands*`)`	gen	color drawing for PSTricks, pgf, MetaPost postprocessors
`rgbfill(`*color triple*`, `*closed path*`)`	gen	fill with arbitrary color
`right_`	gen	set current direction right
`rjust_`	gen	right justify with respect to current direction
`rlabel(`*label*`,`*label*`,`*label*`)`	cct	triple label 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
`rotbox(`*wid,ht,type*`)`	gen	box oriented in current direction in `[ ]` block; *type*= e.g. `dotted shaded "green"`. Defined internal locations: N, NE, E, SE, S, SW, W, NW.
`rotellipse(`*wid,ht,type*`)`	gen	ellipse oriented in current direction in `[ ]` block; e.g. `Point_(45); rotellipse(,,dotted fill_(0.9)).` Defined internal locations: N, S, E, W.
`round(`*line thickness*`,at `*location*`,`*color*`)`	gen	filled circle for rounded corners
`rpoint_(`*linespec*`)`	gen	set direction cosines

`rpos_`(*position*)	gen	Here + *position*
`rrot_`(*x, y, angle*)	gen	`Here + vrot_`(*x, y, cos(angle), sin(angle))*
`rs_box(`*text*`,`*expr1*`,`···`)`	gen	like `s_box` but the text is rotated by `text_ang` (default 90) degrees
`rsvec_`(*position*)	gen	Here + *position*
`rt_`	gen	right with respect to current direction
`rtod_`	gen	constant, degrees/radian
`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*`,`···`)`	gen	generate dimensioned text string using `\boxdims` from `boxdims.sty`. Two or more args are passed to `sprintf()`
`s_dp(`*name*`,`*default*`)`	gen	depth of the most recent (or named) `s_box`
`s_ht(`*name*`,`*default*`)`	gen	height of the most recent (or named) `s_box`
`s_init(`*name*`)`	gen	initialize `s_box` string label to *name* which should be unique
`s_name`	gen	the value of the last `s_init` argument
`s_wd(`*name*`,`*default*`)`	gen	width of the most recent (or named) `s_box`
`sc_draw(`*dna string, chars, iftrue, iffalse*`)`	cct	test if chars are in string, deleting chars from string
`se_`	gen	.se with respect to current direction
`setrgb(`*red value, green value, blue value*`,[`*name*`])`	gen	define colour for lines and text, optionally named (default `lcspec`)
`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
`sfgarc(`*linespec*`,`*text*`,`*text justification*`,cw|ccw,` *height scale factor*`)`	cct	directed arc drawn between nodes, with text label and a height-adjustment parameter
`sfgbelow`	cct	like below but with extra space
`sfgline(`*linespec*`,`*text*`,`*text justification*`)`	cct	directed straight line chopped by node radius, with text label
`sfgnode(at` *location*`,`*text*`,above|below,sl circle options)`	cct	small circle default white interior, with text label. The default label position is inside if the diameter is bigger than `textht` and `textwid`; otherwise it is `sfgabove.` Options such as fill or line thickness can be given.
`sfgself(at` *location*`, U|D|L|R|`*degrees*`,` *text*`,` *text justification*`, cw|ccw,` *scale factor*`)`	cct	self-loop drawn at angle *angle* from a node, with text label and a size-adjustment parameter
`shade(`*gray value*`,`*closed line specs*`)`	gen	fill arbitrary closed curve
`shadebox(`*box specification*`)`	gen	box with edge shading

`sign_(`*number*`)`	gen	sign function
`sinc(`*number*`)`	gen	the sinc(x) function
`sind(`*arg*`)`	gen	sine of an expression in degrees
`sinusoid(`*amplitude, frequency, phase, tmin, tmax*`)`	gen	draws a sinusoid over the interval ($t_{\text{min, max}}$
`source(`*linespec*`,V|v|I|i|AC|F|G|Q|L|P|S|T|X|U|`*string*`,`*diameter*`,R)`	cct	source, blank or voltage (2 types), current (2 types), AC, or type F, G, Q, L, X or labelled, P = pulse, U = square, R = ramp, S = sinusoid, T = triangle; R = reversed polarity
`sourcerad_`	cct	default source radius
`sp_`	gen	evaluates to medium space for gpic strings
`speaker( U|D|L|R|`*degrees*`,`*size*`,H)`	cct	speaker, *In1* to *In7* defined; `H`=horn
`sprod3D(a,x,y,z)`	3D	scalar product of triple x,y,z by a
`sum3D(x1,y1,z1,x2,y2,z2)`	3D	sum of two triples
`sum_(`*a*`,`*b*`)`	gen	binary sum
`svec_(`*x*`,`*y*`)`	log	scaled and rotated grid coordinate vector
`sw_`	gen	.sw with respect to current direction
`switch(`*linespec*`,L|R,[C|O][D],[B|D])`	cct	SPST switch (wrapper for bswitch, lswitch, and dswitch macros), arg2: R=right orientation (default L=left); if arg4=blank (knife switch): arg3 = [O|C][D] O= opening, C=closing, D=dots; if arg4=B (button switch): arg3 = [O|C] O=normally open, C=normally closed, if arg4=D: arg3 = same as for dswitch
`ta_xy(`*x, y*`)`	cct	macro-internal coordinates adjusted for `L|R`
`tgate(`*linespec,* `[B][R|L])`	cct	transmission gate, `B=` ebox type; `L=` oriented left
`thicklines_(`*number*`)`	gen	set line thickness in points
`thinlines_(`*number*`)`	gen	set line thickness in points
`thyristor(`*linespec*`,`*chars*`)`	cct	thyristor, chars: D=Diode, A=Open diode, UA=open diode with centre line, B=Bidirectional diode, C=Type IEC, G=Full-size gate terminal, R=Right orientation, E=envelope
`tline(`*linespec*`,`*wid*`,`*ht*`)`	cct	transmission line, manhattan direction
`tr_xy(`*x, y*`)`	cct	relative macro internal coordinates adjusted for `L|R`
`tr_xy_init(`*origin, unit size, sign* `)`	cct	initialize `tr_xy`
`transformer(`*linespec*`,L|R,np,[A][W|L],ns)`	cct	2-winding transformer: left or right, np primary arcs, air core or wide or looped windings, ns secondary arcs
`ttmotor(`*linespec, string, diameter, brushwid, brushht*`)`	cct	motor with label
`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
`variable(`‘*element*’`, [A|P|L|[u]N][C|S],`*angle*`,`*length*`)`	cct	overlaid arrow or line to indicate variable 2-terminal element: `A`=arrow, `P`=preset, `L`=linear, `N`=nonlinear, `C`=continuous, `S`=setpwise
`vec_(`*x*`,`*y*`)`	gen	position rotated with respect to current direction
`vlength(`*x*`,`*y*`)`	gen	vector length $\sqrt{x^2+y^2}$
`vperp(`*linear object*`)`	gen	unit-vector pair CCW-perpendicular to linear object
`vrot_(`*x*`,`*y*`,`*xcosine*`,`*ycosine*`)`	gen	rotation operator
`vscal_(`*number*`,`*x*`,`*y*`)`	gen	vector scale operator
`w_`	gen	.w with respect to current direction
`wid_`	gen	width with respect to current direction
`winding(L|R,` *diam, pitch, turns, core wid, core color*`)`	cct	core winding drawn in the current direction; `R`=right-handed
`xtal(`*linespec*`)`	cct	quartz crystal
`xtract(`*string, substring*`)`	gen	returns substring if present

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] D. Girou. Présentation de PSTricks. *Cahiers GUTenberg*, 16, 1994. `http://www.gutenberg.eu.org/pub/GUTenberg/publicationsPDF/16-girou.pdf`.

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

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

[7] KDE-Apps.org. Cirkuit, 2009. KDE application: `http://kde-apps.org/content/show.php/Cirkuit?content=107098`.

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

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

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

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

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

[13] R. Seindal *et al.* GNU m4, 1994. `http://www.gnu.org/software/m4/manual/m4.html`.

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