M4 Macros for Electric Circuit Diagrams in LaTeX Documents

Dwight Aplevich

Version 5.97

Contents

1 Introduction

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

This document describes a set of macros, written in the **m4** macro language [7], for producing electric circuits and other diagrams in LaTeX documents. The macros evaluate to drawing commands in **pic**, a line-drawing language [8] that is readily available and quite simple to learn. The result is a system with the advantages and disadvantages of TeX itself, since it is macro-based and non-wysiwyg, and since it uses ordinary character input. The book from which the above quotation is taken correctly points out that the payoff 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. For particular drawings it is often convenient to customize elements or to package combinations of them, so macros such as these are only a starting point. The IEEE standard [6] has been followed most of the time. The macros described here make extensive use of the characteristics of **pic** and have been designed, where possible, to be an extension of the language.

2 Using the macros

The diagram source file is preprocessed as illustrated in 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.

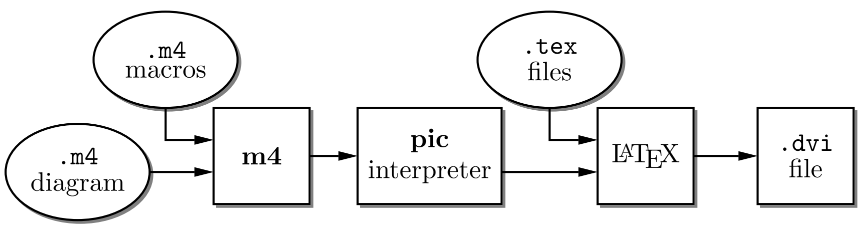

Figure 1: Inclusion of figures and macros in the LaTeX document. Replacing LaTeX with PDFlatex to produce pdf directly is also possible.

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

There are two principal choices of **pic** interpreter. One is [3] **gpic -t** together with a printer driver that understands **tpic** specials, typically [11] **dvips**. In some installations, **gpic** is simply named **pic,** but make sure that GNU **pic** [3] is being invoked rather than the older Unix **pic.** An alternative is **dpic**, described later in this document. **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. The fastidious reader might observe that there are three languages being scrambled: **m4**, **pic**, and the **tpic, tex** or other output, not to mention the meta-language of the macros, and that this mixture might be a problem, but experience implies otherwise.

2.1 Quick start

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

```
.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
```

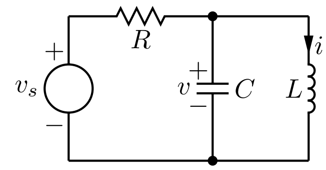

Figure 2: The file `quick.m4` and resulting diagram.

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

2.1.1 Processing with gpic

If your printer driver understands **tpic** specials and you are using **gpic** (on some systems the **gpic** command is `pic`), do the following. Type

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

where `<path>` is the path to the `libcct.m4` file. Add the following to your main LaTeX source file:

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

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

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

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

and the main LaTeX source file should have the statement `\usepackage{pstricks}` in the header. The figure inclusion statements are

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

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 `-g` **dpic** option as follows:

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

The 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.

2.1.3 Simplifications

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

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

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

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

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

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

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

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

When many files are to be processed, then a facility such as Unix **make**, which is also available in several PC versions, can be employed to automate the manual commands given above. On systems without such a facility, a scripting language can be used. Alternatively, you can put several diagrams into a single source file so that they can be processed together, as follows. 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.

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 [8] can be obtained from URL:

`http://www.cs.bell-labs.com/10thEdMan/pic.pdf`

A more complete manual [10] is included in the GNU **groff** package. A compressed postscript versions 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 [4] 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 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. 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 and 1 meaning respectively black and 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 way of occurrence associates 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 group of statements enclosed in square brackets is a compound object. 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 elements

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

Another possibility for defining positions is

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

which is abbreviated as

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

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

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

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

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

The logical operators `==, !=, <=, >=, >, <` apply to expressions, and strings can be tested for equality or inequality. A modest selection of numerical functions is also provided: the single-argument functions `sin, cos, log, exp, sqrt, int`, where `log` and `exp` are base-10, the 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 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** command; the first part of the source is as follows:

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

R1: resistor
```

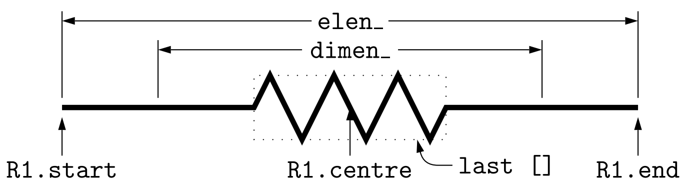

Figure 3: Resistor named `R1`, showing the size parameters, enclosing block, and predefined positions.

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

- The first line invokes an almost-empty macro 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** environment variable with default value 0.5 in. The default length of an element is `elen_`, which is `dimen_*3/2`. 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).
- The two-terminal element macros expand to sequences of drawing commands that begin with '`line invis` *linespec*', where *linespec* is the first argument of the macro if it is non-blank, otherwise by default the line is drawn a distance `elen_` in the current direction, which is to the right by default. The invisible line is first drawn, then the element is drawn on top of the line. The element—rather the initially-drawn invisible line—can be given a name, `R1` in the example, so that positions `R1.start`, `R1.centre`, and `R1.end` are defined as shown.
- The element body is enclosed by a block, which later can be used to place labels around the element. The block corresponds to an invisible rectangle with horizontal top and bottom lines, regardless of the direction in which the element is drawn. In the diagram a dotted box has been drawn to show the block boundaries.
- The last sub-element, identical to the first in each two-terminal element, is an invisible line that can be referenced later to place labels or other elements. This 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 0.4pt, and the box displaying the element body block is drawn. Notice how the width and height can be specified, and the box centre positioned at the centre of the block.
- The next paragraph draws two objects, a spline with an arrowhead, and a string left justified at the end of the spline. Other string-positioning modifiers than `ljust` are `rjust, above,` and `below.` Lines to be read by **pic** can be continued by putting a backslash as the rightmost character.
- The last paragraph invokes a macro for dimensioning diagrams.

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. 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 before arguments.

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.

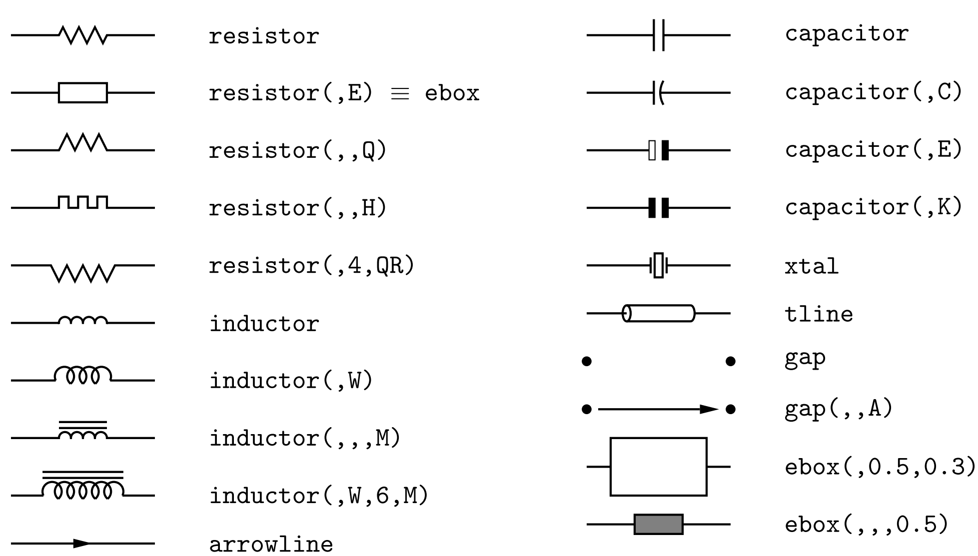

Figure 4: Two-terminal elements, showing some variations.

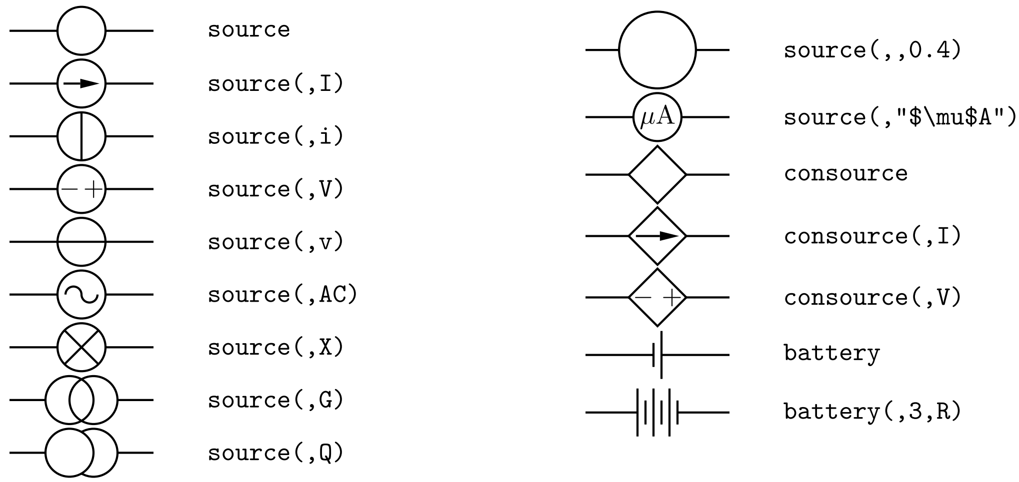

Figure 5: Sources and source-like elements.

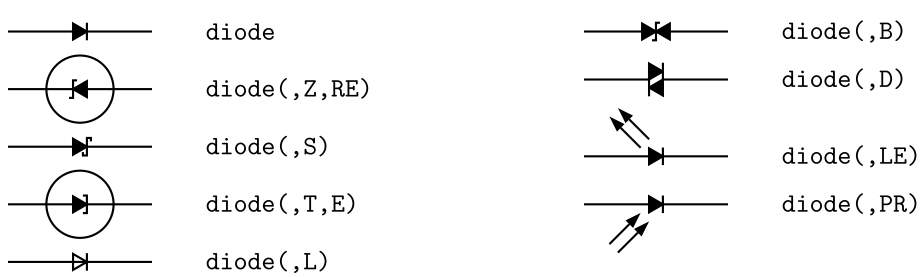

Figure 6: Variants of `diode(`*linespec*`, B|D|L|LE[R]|P[R]|S|T|Z,[R][E])`.

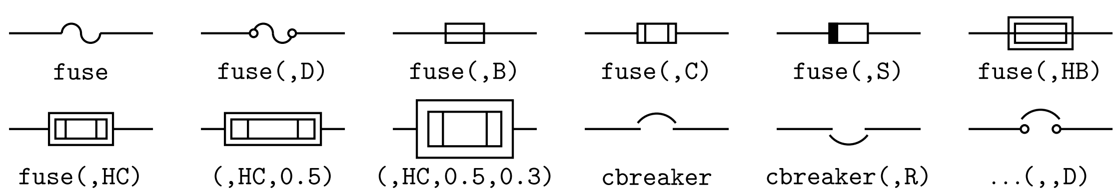

Figure 7: The `fuse(`*linespec*`, A|dA|B|C|D|E|S|HB|HC,` *wid*`,` *ht*`)` and `cbreaker(`*linespec*`,L|R,D)` macros.

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

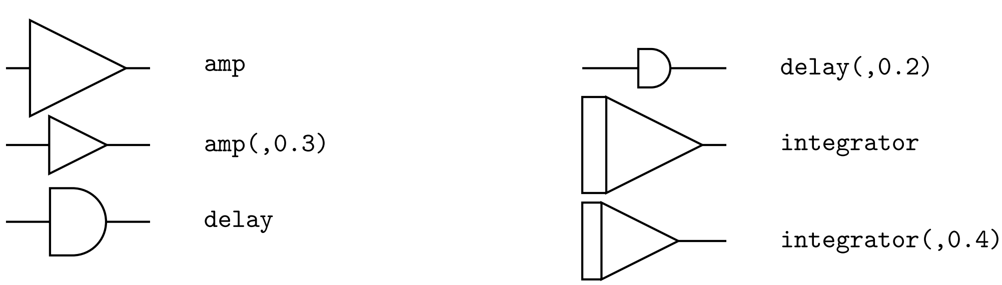

Figure 8: Amplifier, delay, and integrator.

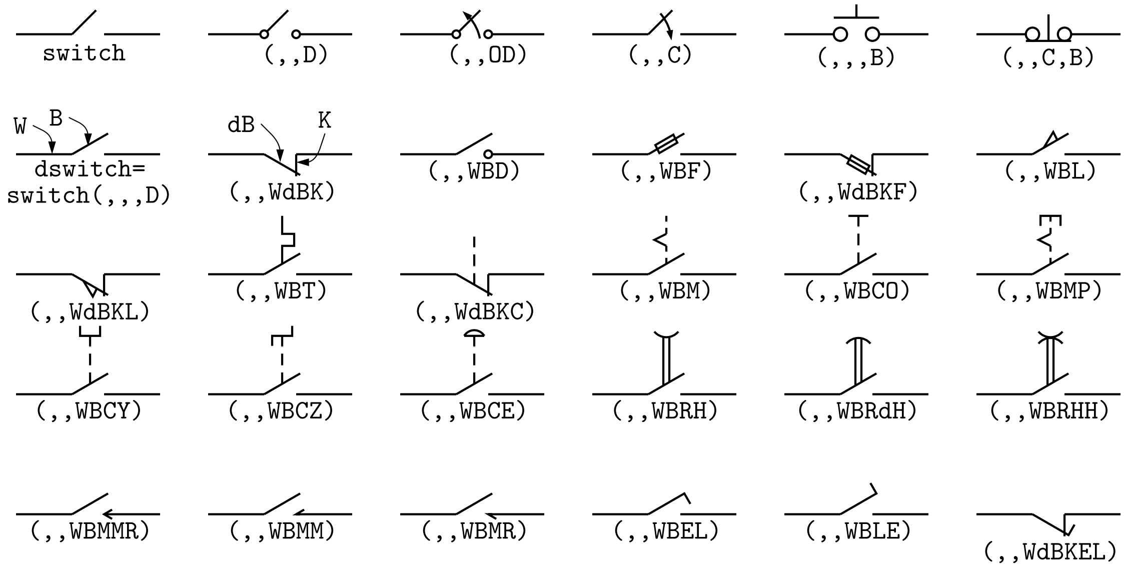

Figure 9: The basic `switch(`*linespec*`,L|R,[O|C][D],B)` and more elabourate `dswitch(`*linespec*`,R,W[ud]B[K]`*chars*`)` macros, with drawing direction `right_`. Setting the second argument to `R` produces a mirror image with respect to the drawing direction. The macro `switch(,,,D)` is a wrapper for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

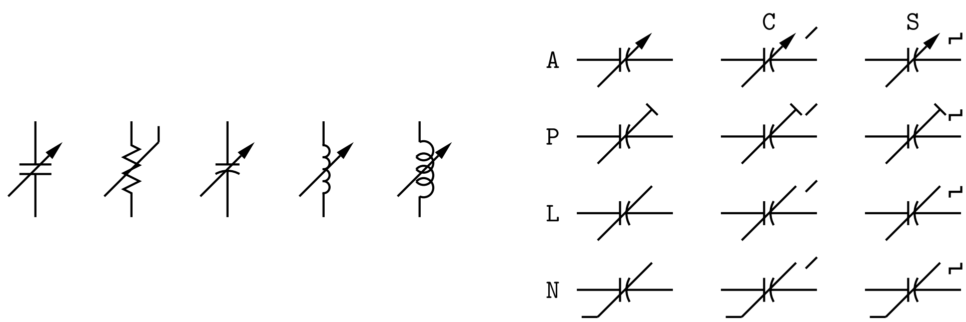

Figure 10: Illustrating `variable(‘`*element*`’,[A|P|L|[u]N][C|S],`*angle*`,`*length*`)`. For example, `variable(‘capacitor(down_ dimen_)’)` draws the leftmost capacitor shown above, and `variable(‘resistor(down_ dimen_)’,uN)` draws the resistor. The default angle is 45°, regardless of the direction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

is drawn with reversed direction. Thus,

```
diode(right_ 0.4); reversed(‘diode’,right_ 0.4)
```

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

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

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

draws the resistor in Figure 10.

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).

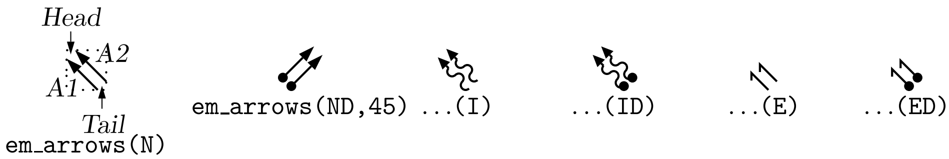

Figure 11: Radiation arrows: `em_arrows(`*type*, *angle*, *length*`)`

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.

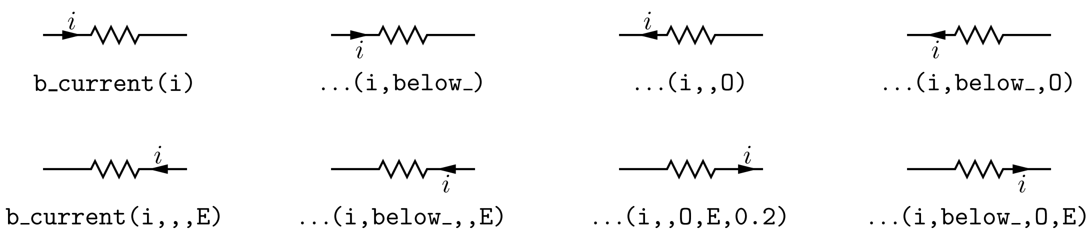

Figure 12: Illustrating `b_current`. In all cases the drawing direction is to the right.

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.

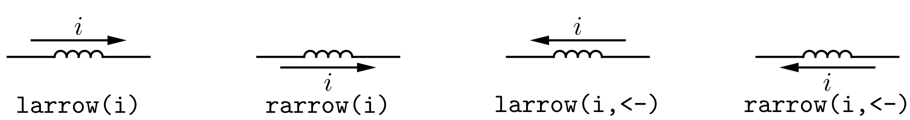

Figure 13: The `larrow` and `rarrow` macros are drawn adjacent to the element to provide a reference direction.

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* `)`

```
% ‘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
```

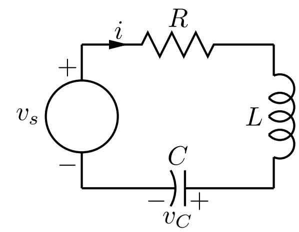

Figure 14: A loop containing labeled elements, with its source code.

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.

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*`,`$\cdots$`)`, 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.

`potentiometer(down_ dimen_)`   `...(down_ dimen_,,0.25,-0.2,0.75,0.2)`

Figure 15: Default and multiple-tap potentiometer.

The ground symbol, shown in Figure 16, has four arguments:
`ground( at` *position*`, T, N|F|S|L|P|E, U|D|L|R|angle )`
so that, for example, the lines

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

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

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 18 illustrates the macro
`opamp(linespec, - label, + label, size, [R][P])`

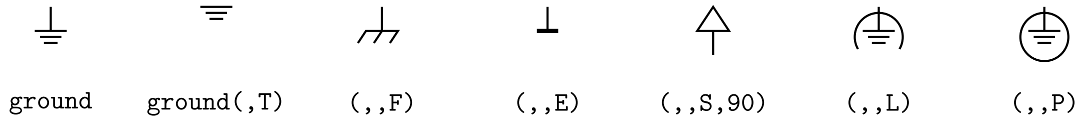

Figure 16: Ground symbols.

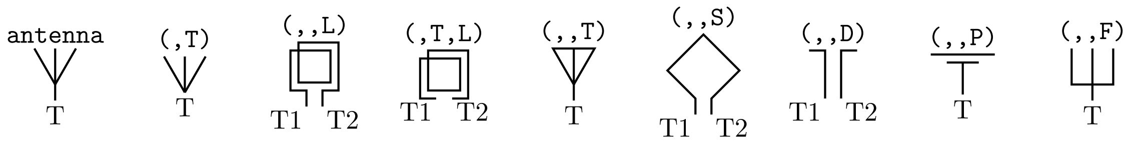

Figure 17: Antenna symbols, with macro arguments shown above and predefined terminal names below.

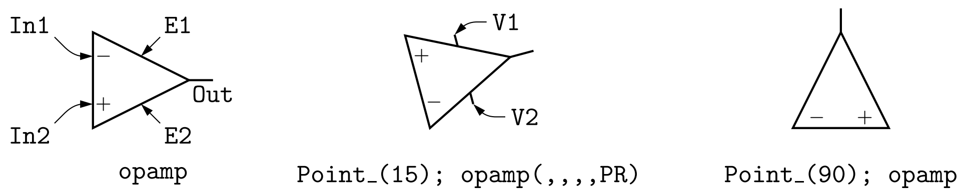

Figure 18: Operational amplifiers. The `P` option adds power connections. The second and third arguments can be used to place and rotate arbitrary text at `In1` and `In2`.

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 adds a power connection, exchanges the second and third entries, or both.

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 **PSTricks** `\rput` commands as second and third arguments, for example.

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

```
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
```

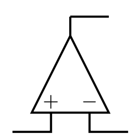

Figure 19: A code fragment invoking the `opamp(`*linespec*`,-,+,`*size*`,[R][P])` macro.

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 argument specifies the number of primary arcs, the fourth omits the iron core, and the fifth specifies the number of secondary arcs.

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.

The seven-argument `nport` 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

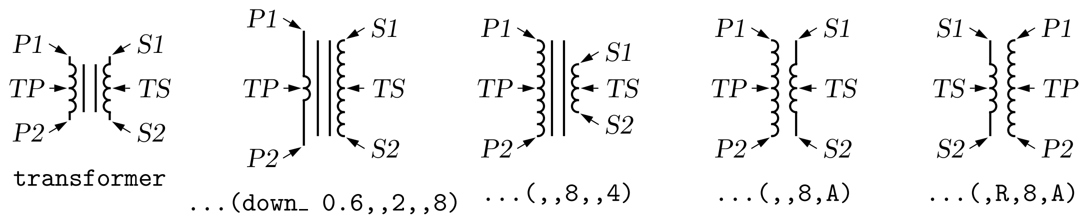

Figure 20: The `transformer(`*linespec*`,L|R,`*np*`,A,`*ns*`)` macro (drawing direction `down`), showing predefined terminal and centre-tap points.

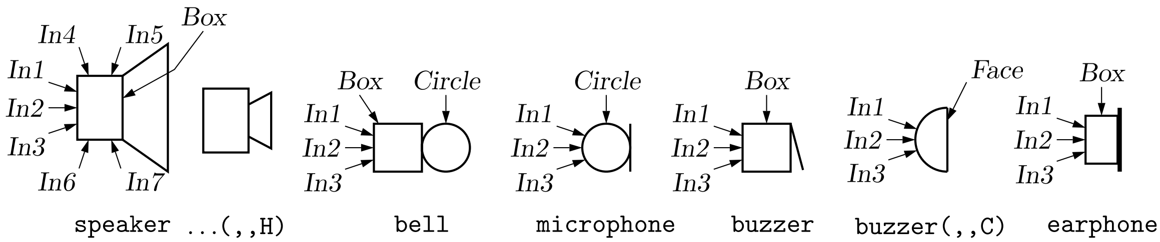

Figure 21: Audio components: `speaker(U|D|L|R|`*degrees*`,`*size*`,`*type*`), bell, microphone, buzzer, earphone`, with their internally named positions and components.

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 last argument to `N` omits the pin dots. The complete structure is enclosed in a block.

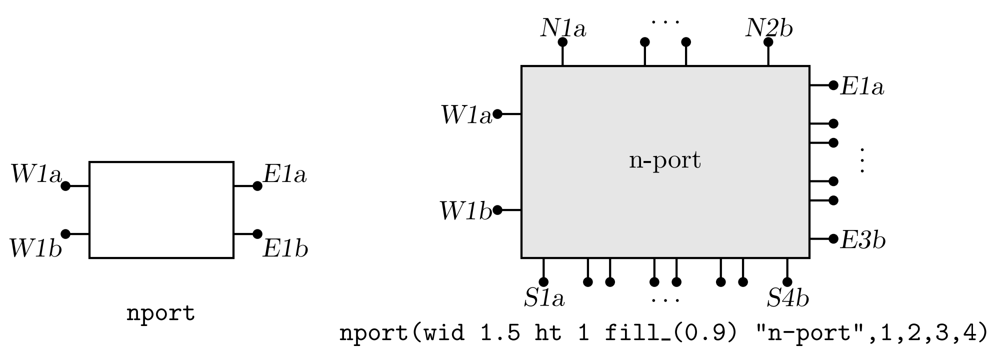

Figure 22: The `nport` macro draws a sequence of pairs of named pins on each side of a box. The default is a twoport. The pin names are shown.

A basic winding macro for magnetic-circuit sketches and similar figures is shown in Figure 23. 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 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 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

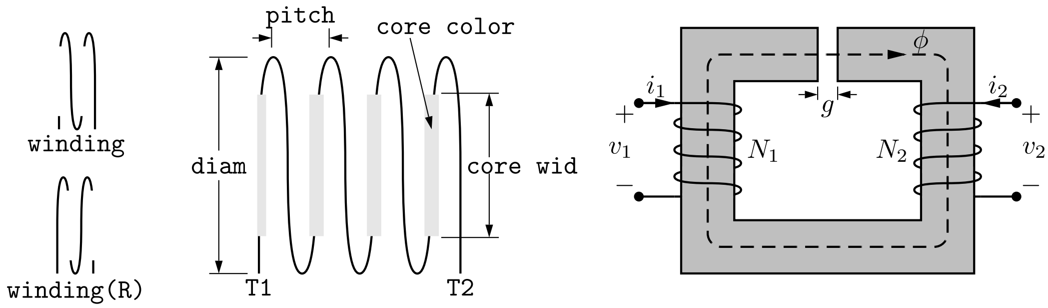

Figure 23: The `winding(L|R, diam, pitch, turns, core wid, core color)` macro draws a coil with axis along the current drawing direction. Terminals `T1` and `T2` are defined. Setting the first argument to `R` draws a right-hand winding.

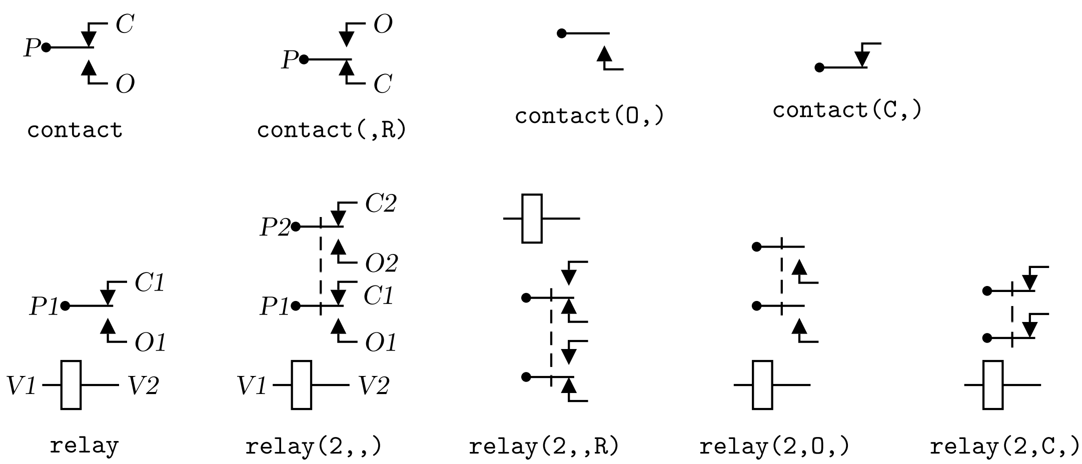

Figure 24: The `contact(O|C,R)` and `relay(`*poles*`,O|C,R)` macros (default direction right).

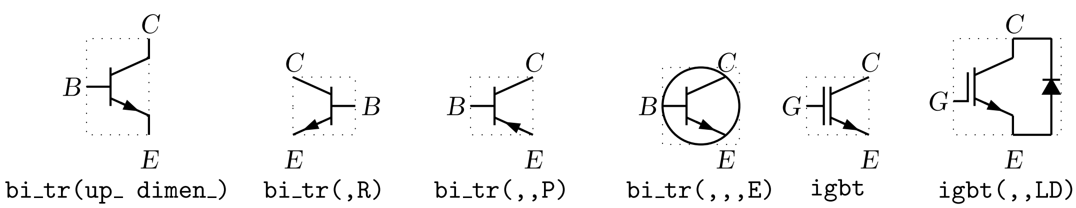

Figure 25: Bipolar transistor variants (current direction upward).

'`E`' draws an envelope around the device. Thus for example, the code fragment in Figure 26 places a bipolar transistor, connects a ground to the emitter, and connects a resistor to the collector.

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 27.

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 28. 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 {`BDEFGLQRSTZ`}, 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 28.

A UJT macro with predefined internal locations *B1*, *B2*, and *E* is illustrated in Figure 29, and an SCR macro with predefined internal locations *T1*, *T2*, and *G* is illustrated in Figure 30. The number

```
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
```

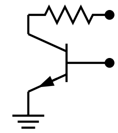

Figure 26: The bi_tr(*linespec*,L|R,P,E) macro.

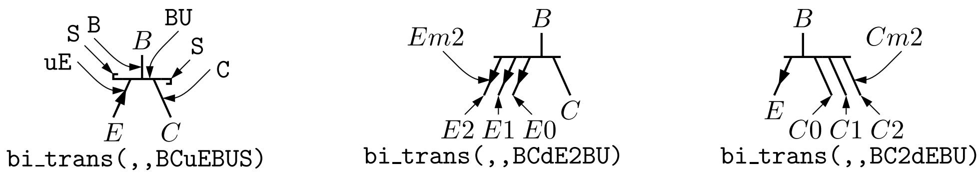

Figure 27: The bi_trans(*linespec*,L|R,*chars*,E) macro. The sub-elements are specified by the third argument. The substring E*n* creates multiple emitters *E0* to *En*. Collectors are similar.

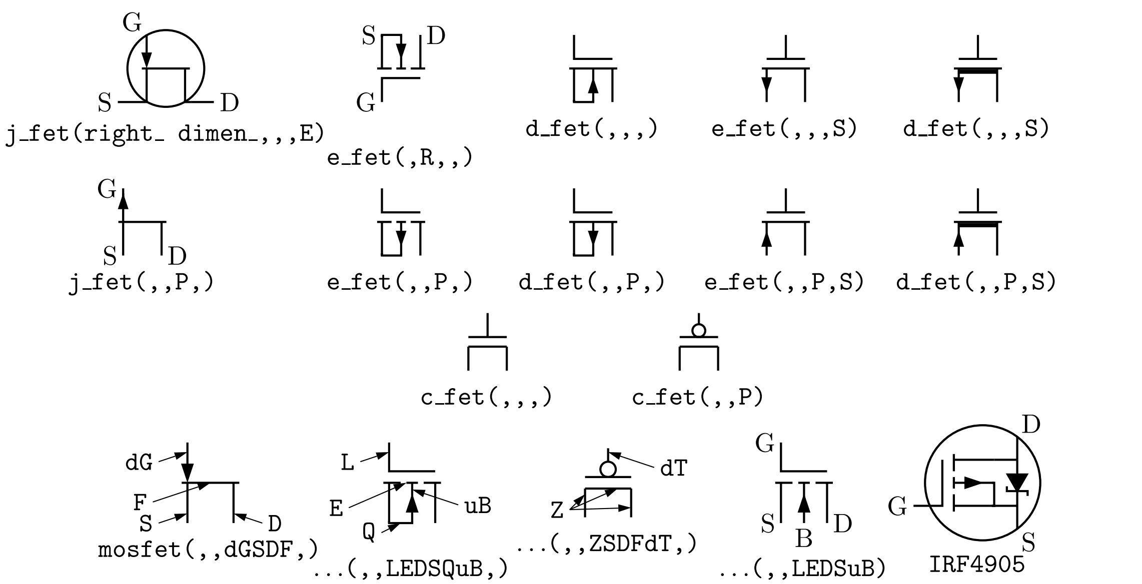

Figure 28: JFET, insulated-gate enhancement and depletion MOSFETS, and simplified versions, see [12]. These macros are wrappers that invoke the mosfet macro as shown in the bottom row. The two lower-right examples show custom devices, the first defined by omitting the substrate connection, and the second defined using a wrapper and custom envelope.

of possible semiconductor symbols is very large, so these macros must be regarded as prototypes. Some other non-two-terminal macros are dot, which has an optional argument 'at *location*', the line-thickness macros, the fill_ macro, and crossover, which is a useful if archaic method to show non-touching conductor crossovers, as in Figure 31.

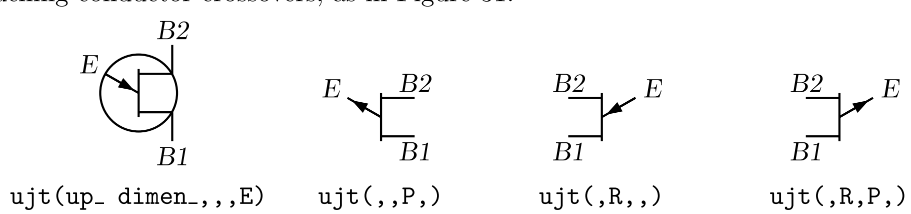

Figure 29: UJT devices, with current drawing direction up.

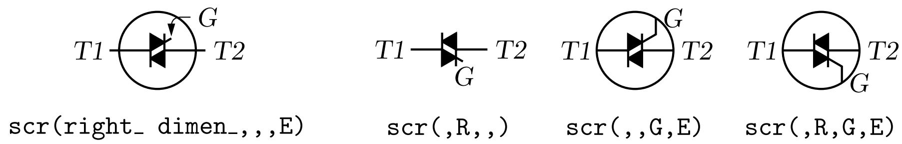

Figure 30: SCR elements, drawing direction to the right.

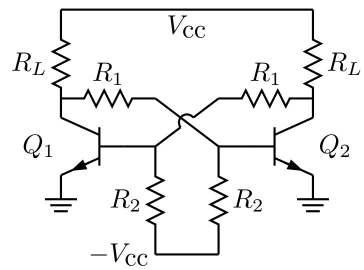

Figure 31: Bipolar transistor circuit, illustrating `crossover`.

6 Directions 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. Thus as shown in Figure 32, '`Point_(-30); resistor`' draws a resistor along a line with slope -30 degrees, and '`rpoint_(to Z)`' sets the current direction cosines to point to location Z. Macro

```
% '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-(Cos(30)*(elen_),0),R) )

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

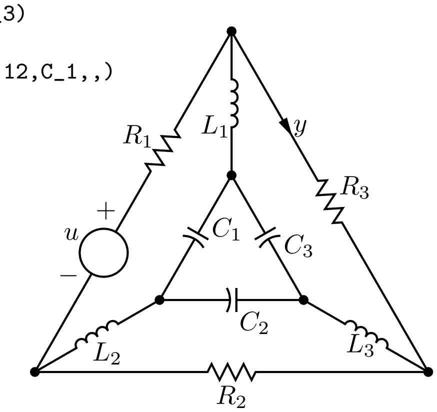

Figure 32: Illustrating elements drawn at oblique angles.

vec_(x,y) evaluates to the position (x,y) rotated by the argument of the previous Point_, point_ or rpoint_ command. The macro rvec_(x,y) evaluates to position Here + vec_(x,y) and is the principal device used to define relative locations in the circuit macros. Thus, line to rvec_(x,0) draws a line of length x in the current direction.

macros. The source for the figure is shown, and illustrates that some hand-placement of labels using dlabel may be useful when elements are drawn obliquely. Because **m4** macro arguments are separated by commas, any commas that are integral parts of the arguments must be protected, either by parentheses as illustrated in inductor(from Cr to Cr+vec_(elen_,0)), or by multiple single quotes, ``,'', as necessary. Commas also may be avoided by writing 0.5 between L and T instead of 0.5<L,T>.

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 purpose. 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, ...

7 Logic gates

Figure 33 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, $\cdots$ InN, 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 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.

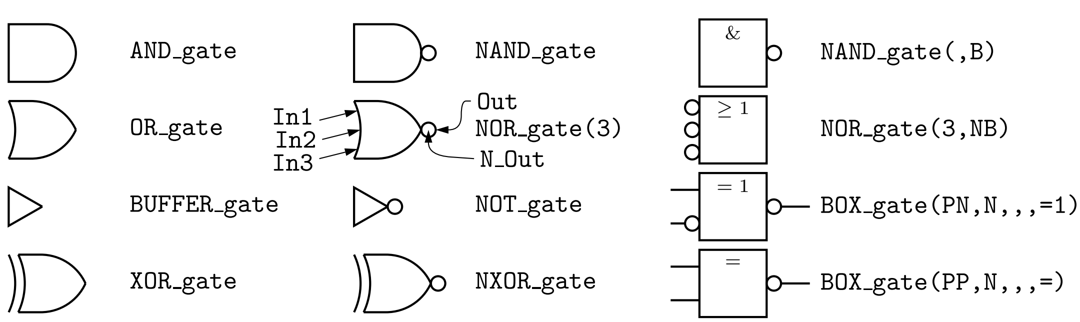

Figure 33: Basic logic gates. The input and output locations of a three-input NOR gate are shown. Inputs are negated by including an N in the second argument letter sequence. A B in the second argument produces a box shape as shown in the rightmost column, where the second example has AND functionality and the bottom two are examples of exclusive OR functions.

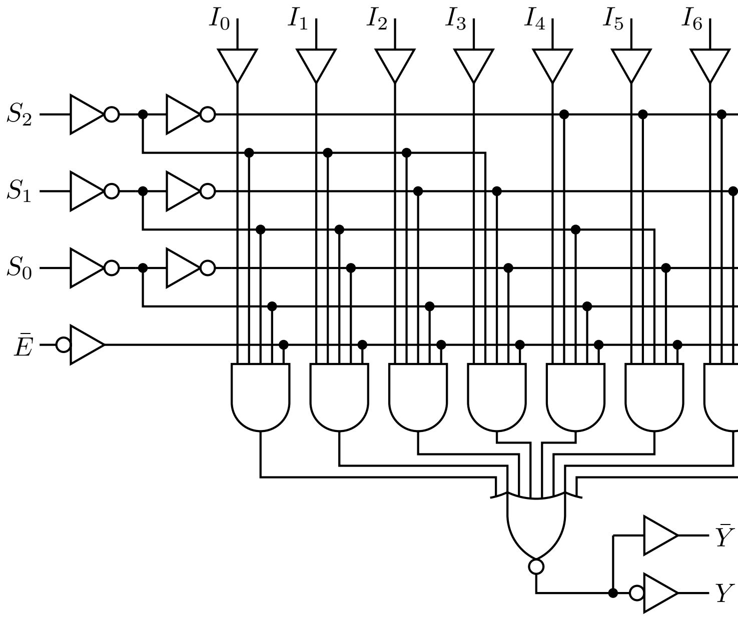

Figure 34: Eight-input binary multiplexer circuit, illustrating a gate with wings and `for_` looping in the source.

Beyond a default number (6) of inputs, the gates are given wings as illustrated in Figure 34.

Input locations retain their positions relative to the gate body regardless of gate orientation, as illustrated in Figure 35.

```
% ‘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
```

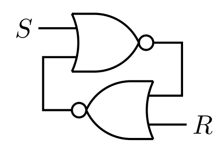

Figure 35: SR flip-flop.

Figure 36 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.

Explicitly defining customized gates is sometimes the simplest technique. For example, the following code defines the custom flip-flops in Figure 37.

```
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)’)
    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)
```

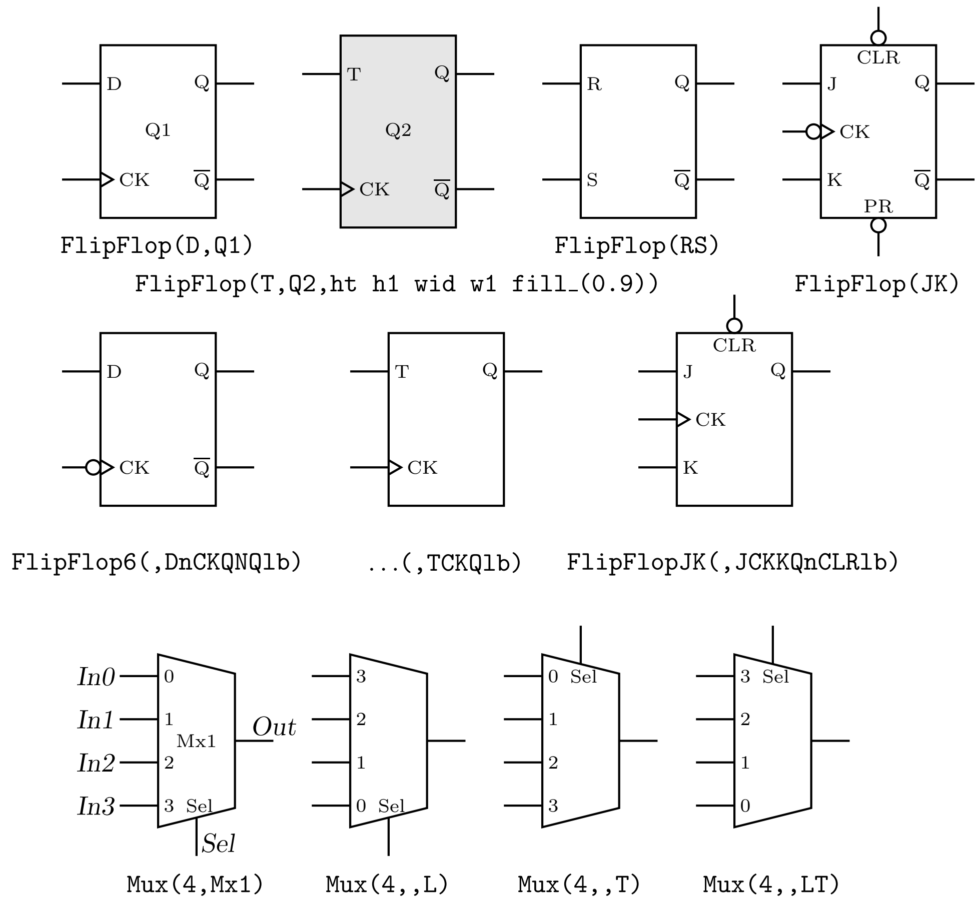

Figure 36: The `FlipFlop` and `Mux` macros, with variations.

```
    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 $\overline{\mathtt{Q}}$ is drawn only if the macro

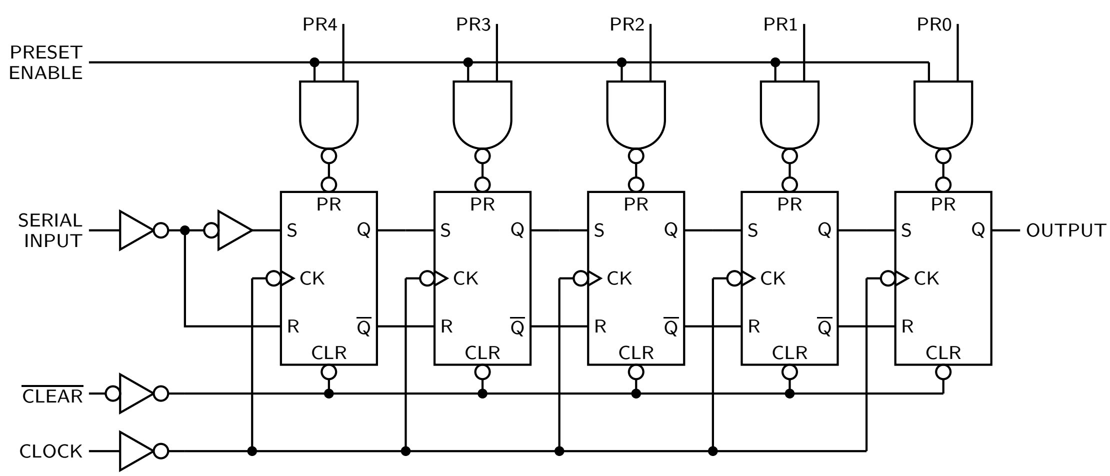

Figure 37: A 5-bit shift register.

argument is 1.

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

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

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

elen = 19                     # Variables are allowed
...
```

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. However, this method is not normally suitable for circuits because arrowheads, line widths, and text are treated differently.

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

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

3. Line widths are independent of diagram and text scaling, and have to be set independently. For example, the assignment `linethick = 1.2` sets the default line width to 1.2 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 [7] or in later manuals [13]. If a new element is required, then modifying and renaming one of the library definitions or simply adding an option to it may suffice. Hints for drawing general two-terminal elements are given in `libcct.m4`. However, if an element or composite is to be drawn in only one orientation then most of the elaborations used for general two-terminal elements in Section 4 can be dropped.

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

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

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

In the following example, two macros are defined to simplify the repeated drawing of a series resistor and series inductor, and the macro `tsection` defines a subcircuit that is replicated several times to generate Figure 38.

```
% ‘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
```

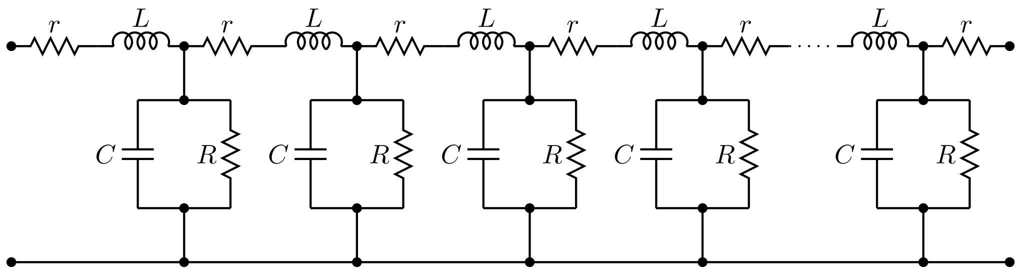

Figure 38: A lumped model of a transmission line, illustrating the use of custom macros.

```
  { 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
```

10 Interaction with LaTeX

The sizes of typeset labels and other TeX boxes are generally unknown prior to the processing of a 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 determine 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.

The solution to this difficulty is to process the diagram twice. The diagram source is processed by **m4** and a **pic** processor, and the main document source is LaTeXed to input the diagram and 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. A summary of this hackery follows:

- Put `\usepackage{boxdims}` into the document source.
- 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; alternatively, invoke the macros `\boxdims` and `boxdim` described below.

The macro `s_box(‘`*text*`’)` evaluates 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 then they are processed by `sprintf`.

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 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).

Recent versions of `boxdims.sty` include the macro

`\boxdimfile{`*dimension file*`}`

for specifying 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* `...`

Figure 39 illustrates the effect of changing the previous example to use `s_box`.

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

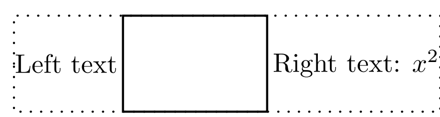

Figure 39: The macro **s_box** sets string dimensions automatically when processed twice. When two or more arguments are present, they are passed through `sprintf`. The dots show the figure bounding box.

Objects can be tailored to their attached text by invoking `\boxdims` and `boxdim` explicitly. The small source file in Figure 40, for example, produces the box in the figure.

```
% ‘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
```

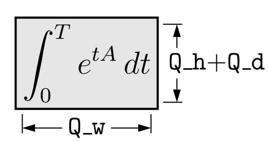

Figure 40: Fitting a box to typeset text.

The source file for the figure is processed by **m4** and a **pic** interpreter to produce a `.tex` file, then LaTeX is run, and then these steps are repeated. 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 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.

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 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.

More tricks can be played. The example

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

shows a nice way of including eps graphics in a diagram. The included picture (named S 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 tricks

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

The rotation of text is illustrated by the file

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

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

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

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

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

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

`command "`\psset{`*option*=*value*, $\cdots$`}'"`

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

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. 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.

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 41 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.

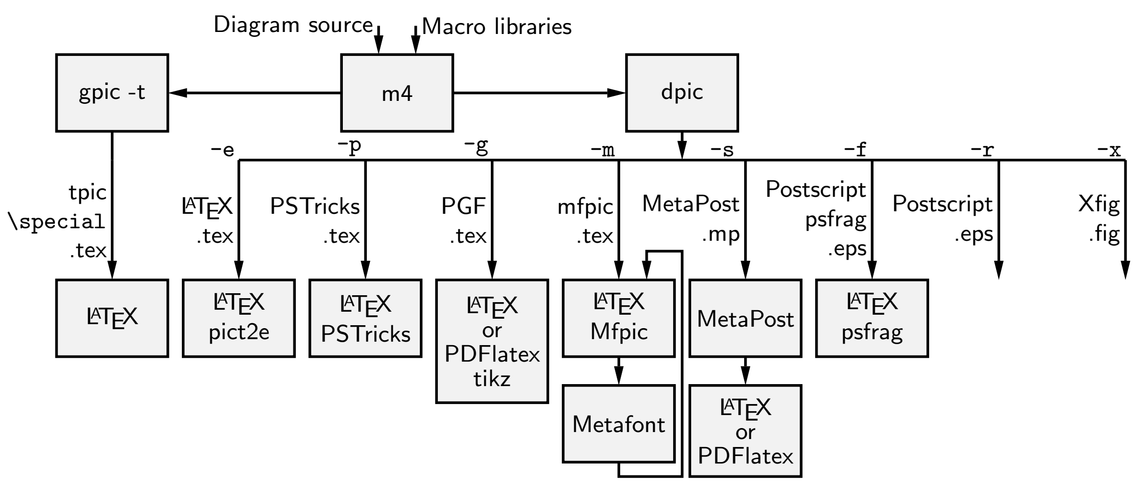

Figure 41: Output formats produced by **gpic** `-t` and **dpic**.

13 Developer's notes

Several years ago in the course of writing a book, I took a few days off to write a **pic**-like interpreter **(dpic)** to automate the tedious coordinate calculations required by LaTeX picture objects. The macros in this distribution and the interpreter are the result of that effort and of drawings I have had to produce since. The interpreter has been upgraded over time to generate **mfpic**, **MetaPost** [5], raw **Postscript**, **Postscript** with **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. In addition, **xfig**-compatible output has been added and, most recently, **Ti*k*Z PGF** output, which combines the simplicity of **PSTricks** with PDFlatex compatibility. 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 the geometrical calculations that are necessary for precision in line drawings. The language is also simple to learn and, more importantly, to read. There are built-in looping and block-structure constructs that combine power with simplicity, and the language has stood the test of time. However, no choice of tool is without compromise, and making good graphics is time-consuming no matter how it is done.

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**. **Dpic** will also issue low-level **MetaPost** or **Postscript** commands, 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®. The user is responsible for ensuring that the correct fonts are provided and for reformatting labels.

14 Bugs

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

In response to suggestions, some of the macros have been modified to allow easier customization to forms not originally anticipated, but this process is not complete.

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

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

   `there is no variable 'rp_ang'`

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

   `cct_init`

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

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

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

   when it should be

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

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

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

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

produces an error, whereas

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

and

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

do not.

4. **Default lengths:** Remember that the *linespec* argument of element macros requires both a direction and a length. Writing

   ```
   source(up_)
   ```

   draws a source up a distance equal to the current `lineht` value, which may cause confusion. It is usually better to specify both the direction and length of an element, thus:

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

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

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

   will be typeset as if it were

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

   The cure is to add single quotes.

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

6. **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$"`.

7. **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}
   ```

8. **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.

9. **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)`

10. **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.

11. **Incompatible processors:** If you switch between **dpic** and **gpic,** remember that the libraries are set up for **gpic** by default, otherwise `pstricks.m4` or one of the other configuration libraries has to be processed before the other libraries. To redefine the default behaviour, change the `include` statements near the top of the libraries.

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

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 example sources contain additional macros, such as for flowcharts and binary trees.

Internal macros defined within the libraries begin with the characters m4 or M4, and 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
`AND_wd`	log	width of basic 'and' and 'or' gates
`BOX_gate(`***inputs,output,swid,sht,label***`)`	log	output=[`P`|`N`], inputs=[`P`|`N`]..., sizes swid and sht in `L_units`

<table>
<tr><td>BUFFER_gate(linespec, N)</td><td>log</td><td>basic buffer, 1 input or as a 2-terminal element, N=negated input</td></tr>
<tr><td>BUFFER_gen(chars,wd,ht,[N|P]*,[N|P]*,[N|P]*)</td><td>log</td><td>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</td></tr>
<tr><td>BUF_ht</td><td>log</td><td>basic buffer gate height</td></tr>
<tr><td>BUF_wd</td><td>log</td><td>basic buffer gate width</td></tr>
<tr><td>Cos(integer)</td><td>gen</td><td>cosine function, integer degrees</td></tr>
<tr><td>E_</td><td>gen</td><td>the constant e</td></tr>
<tr><td>Fector(x1,y1,z1,x2,y2,z2)</td><td>3D</td><td>vector projected on current view plane with top face of 3-dimensonal arrowhead normal to x2,y2,z2</td></tr>
<tr><td>FlipFlop(D|T|RS|JK, label, boxspec)</td><td>log</td><td>flip-flops, boxspec=e.g. ht x wid y</td></tr>
<tr><td>FlipFlop6(label, spec, boxspec)</td><td>log</td><td>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</td></tr>
<tr><td>FlipFlopJK(label, spec,boxspec)</td><td>log</td><td>JK flip-flop, spec similar to above</td></tr>
<tr><td>G_hht_</td><td>log</td><td>gate half-height</td></tr>
<tr><td>HOMELIB_</td><td>all</td><td>directory containing libraries</td></tr>
<tr><td>IOdefs(linespec,label,[P|N]*,L|R)</td><td>log</td><td>Define locations label1, ... labeln along the line; P= label only; N=with NOT_circle; R=circle to right of current direction</td></tr>
<tr><td>Intersect_(Name1,Name2)</td><td>gen</td><td>intersection of two named lines</td></tr>
<tr><td>L_unit</td><td>log</td><td>logic-element grid size</td></tr>
<tr><td>LH_symbol(U|D|L|R|degrees)</td><td>log</td><td>logic-gate hysteresis symbol</td></tr>
<tr><td>LT_symbol(U|D|L|R|degrees)</td><td>log</td><td>logic-gate triangle symbol</td></tr>
<tr><td>Max(arg, arg, ...)</td><td>gen</td><td>Max of an arbitrary number of inputs</td></tr>
<tr><td>Min(arg, arg, ...)</td><td>gen</td><td>Min of an arbitrary number of inputs</td></tr>
<tr><td>Mux(n, label, [L][T])</td><td>gen</td><td>binary multiplexer, n inputs, L reverses pin numbers, T puts Sel pin to top</td></tr>
<tr><td>NAND_gate(n,N)</td><td>log</td><td>‘nand’ gate, 2 or n inputs; N=negated input</td></tr>
<tr><td>NOR_gate(n,N)</td><td>log</td><td>‘nor’ gate, 2 or n inputs; N=negated input</td></tr>
<tr><td>NOT_gate(linespec,N)</td><td>log</td><td>‘not’ gate, 1 input or as a 2-terminal element, N=negated input</td></tr>
<tr><td>NXOR_gate(n,N)</td><td>log</td><td>‘nxor’ gate, 2 or n inputs; N=negated input</td></tr>
<tr><td>NOT_circle</td><td>log</td><td>‘not’ circle</td></tr>
<tr><td>N_diam</td><td>log</td><td>diameter of ‘not’ circles</td></tr>
<tr><td>OR_gate(n,N)</td><td>log</td><td>‘or’ gate, 2 or n inputs; N=negated input</td></tr>
<tr><td>OR_gen(n,chars,[wid,[ht]])</td><td>log</td><td>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</td></tr>
<tr><td>Point_(integer)</td><td>gen</td><td>sets direction cosines in degrees</td></tr>
<tr><td>Rect_(radius,angle)</td><td>gen</td><td>(deg) polar-to-rectangular conversion</td></tr>
<tr><td>Sin(integer)</td><td>gen</td><td>sine function, integer degrees</td></tr>
<tr><td>XOR_gate(n,N)</td><td>log</td><td>‘xor’ gate, 2 or n inputs; N=negated input</td></tr>
<tr><td>above_</td><td>gen</td><td>string position above relative to current direction</td></tr>
<tr><td>abs_(number)</td><td>gen</td><td>absolute value function</td></tr>
<tr><td>amp(linespec,size)</td><td>cct</td><td>amplifier</td></tr>
<tr><td>antenna(at location, T, A|L|T|S|D|P|F, U|D|L|R|degrees)</td><td>cct</td><td>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)</td></tr>
<tr><td>arca(chord linespec, ccw|cw, radius, modifiers)</td><td>gen</td><td>arc with acute angle (obtuse if radius is negative)</td></tr>
<tr><td>arcr(center,radius,start angle,end angle)</td><td></td><td></td></tr>
</table>

	gen	arc definition, e.g., `arc(A,r,0,pi_/2) cw ->`
`arcd(`*center*`,`*radius*`,`*start degrees*`,`*end degrees*`)`	gen	arc definition (see `arcr`), angles in degrees
`arrowline(`*linespec*`)`	cct	line (dotted, dashed permissible) with centred arrowhead
`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`=collectors C0 to Cn; `C`=single collector; `G`=gate line and location; `H`=gate line; `L`=L-gate line and location; `[d]D`=named parallel diode, `d`=dotted connection
`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`)
`buzzer( U|D|L|R|degrees, size,[C])`	cct	buzzer, *In1* to *In3* defined, C=curved
`b_current(`*label*`,`*pos*`,In|Out,Start|End,`*frac*`)`	cct	labelled branch-current arrow to *frac* between branch end and body
`c_fet(`*linespec*`,L|R,P)`	cct	left or right, plain or negated pin simplified MOSFET
`capacitor(`*linespec*`,C,R)`	cct	capacitor, C=curved-plate, E=polarized boxed plates, K=filled boxed plates, R=reversed polarity
`cbreaker(`*linespec*`, L|R, D)`	cct	circuit breaker to left or right, `D`=dotted
`clabel(`*label*`,`*label*`,`*label*`)`	cct	centre triple label
`contact(O|C,R)`	cct	single-pole contact: default double pole or normally open or closed, oriented to the left or right
`contline(`*line*`)`	gen	evaluates to `continue` if processor is **dpic**, otherwise to first arg (default `line`)
`consource(`*linespec*`,V|I)`	cct	voltage or current controlled source
`cosd(`*arg*`)`	gen	cosine of an expression in degrees
`cross(at` *location*`)`	gen	plots a small cross
`cross3D(x1,y1,z1,x2,y2,z2)`	3D	cross product of two triples
`crossover(`*linespec*`, L|R, Line1, ...)`	cct	line jumping left or right over named lines
`crosswd_`	gen	cross dimension
`csdim_`	cct	controlled-source width
`d_fet(`*linespec*`,L|R,P,S,E|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
`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
`delay(`*linespec*`,`*size*`)`	cct	delay element
`delay_rad_`	cct	delay radius
`dend(at` *location*`)`	darrow	close (or start) double line
`diff_(`*a*`,`*b*`)`	gen	difference function

<table>
<tr><td>diff3D(x1,y1,z1,x2,y2,z2)</td><td>3D</td><td>difference of two triples</td></tr>
<tr><td>dimen_</td><td>cct</td><td>size parameter for circuit elements</td></tr>
<tr><td>dimension_(linespec,offset,label,D|H|W|blank width,tic offset,arrowhead)</td><td>gen</td><td>macro for dimensioning diagrams; arrowhead=-> | <-</td></tr>
<tr><td>diode(linespec,B|D|L|LE[R]|P[R]|S|T|Z,[R][E])</td><td>cct</td><td>diode: bi-directional, diac, Schottky, tunnel, zener, LED (right), photodiode (right), open; R=reversed polarity, E=enclosure</td></tr>
<tr><td>direction_(U|D|L|R|degrees, default)</td><td>gen</td><td>sets current direction up, down, left, right, or angle in degrees.</td></tr>
<tr><td>dlabel(long,lat,label,label,label)</td><td>cct</td><td>general triple label</td></tr>
<tr><td>dleft</td><td>darrow</td><td>double line left turn</td></tr>
<tr><td>dline(linespec,t,t,width, |-| or -| or |-)</td><td>darrow</td><td>double line, truncated by half width at either end, closed at either or both ends</td></tr>
<tr><td>dlinewid</td><td>darrow</td><td>width of double lines</td></tr>
<tr><td>dljust(at location)</td><td>darrow</td><td>ljust (displaced dlinewid/2)</td></tr>
<tr><td>dn_</td><td>gen</td><td>sets down relative to current-direction</td></tr>
<tr><td>dot(at location,radius,fill)</td><td>gen</td><td>filled circle (third arg= gray value: 0=black, 1=white)</td></tr>
<tr><td>dot3D(x1,y1,z1,x2,y2,z2)</td><td>3D</td><td>dot product of two triples</td></tr>
<tr><td>dotrad_</td><td>gen</td><td>dot radius</td></tr>
<tr><td>down_</td><td>gen</td><td>sets current direction to down</td></tr>
<tr><td>dright</td><td>darrow</td><td>double arrow right turn</td></tr>
<tr><td>drjust(at location)</td><td>darrow</td><td>rjust (displaced dlinewid/2)</td></tr>
<tr><td>dswitch(linespec,L|R,W[ud]B[K]chars)</td><td>cct</td><td>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 = dotted contact, 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</td></tr>
<tr><td>dtee([L|R])</td><td>darrow</td><td>double arrow tee junction with tail to left, right, or (default) back along current direction</td></tr>
<tr><td>dtor_</td><td>gen</td><td>degrees to radians conversion constant</td></tr>
<tr><td>dturn(degrees ccw)</td><td>darrow</td><td>turn dline arg1 degrees left (ccw)</td></tr>
<tr><td>e_</td><td>gen</td><td>.e relative to current direction</td></tr>
<tr><td>e_fet(linespec,L|R,P,S,E|S)</td><td>cct</td><td>left or right, N or P enhancement MOSFET, normal or simplified, without or with envelope or thick channel</td></tr>
<tr><td>earphone(U|D|L|R|degrees, size)</td><td>cct</td><td>earphone, In1 to In3 defined</td></tr>
<tr><td>ebox(linespec,length,ht,fill value)</td><td>cct</td><td>two-terminal box element with adjustable dimensions and fill value 0 (black) to 1 (white)</td></tr>
<tr><td>eleminit_(linespec)</td><td>cct</td><td>internal line initialization</td></tr>
<tr><td>elen_</td><td>cct</td><td>default element length</td></tr>
<tr><td>em_arrows([N|I|E][D],angle,length)</td><td>cct</td><td>radiation arrows N=nonionizing, I=ionizing, E=simple; D=dot</td></tr>
<tr><td>endshade</td><td>gen</td><td>end gray shading, see beginshade</td></tr>
<tr><td>expe</td><td>gen</td><td>exponential, base e</td></tr>
<tr><td>fill_(number)</td><td>gen</td><td>fill macro, 0=black, 1=white</td></tr>
<tr><td>for_(start,end,increment,‘actions’)</td><td>gen</td><td>integer for loop with index variable m4x</td></tr>
</table>

<table>
<tr><td>fuse(linespec, type, wid, ht)</td><td>cct</td><td>fuse symbol, type= A|B|C|D|S|HB|HC or dA=D</td></tr>
<tr><td>gap(linespec,fill,A)</td><td>cct</td><td>gap with (filled) dots, A=chopped arrow between dots</td></tr>
<tr><td>glabel_</td><td>cct</td><td>internal general labeller</td></tr>
<tr><td>gpar_(element,element,separation)</td><td>cct</td><td>two same-direction elements in parallel</td></tr>
<tr><td>gpic_</td><td>gpic</td><td>defined to signify gpic is being used</td></tr>
<tr><td>grid_(x,y)</td><td>log</td><td>absolute grid location</td></tr>
<tr><td>ground(at location, T, N|F|S|L|P|E, U|D|L|R|degrees)</td><td>cct</td><td>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)</td></tr>
<tr><td>hop(L|R,at location)</td><td>cct</td><td>conductor crossing another to left or right</td></tr>
<tr><td>hoprad_</td><td>cct</td><td>hop radius</td></tr>
<tr><td>ht_</td><td>gen</td><td>height relative to current direction</td></tr>
<tr><td>igbt(linespec,L|R,[L][[d]D])</td><td>cct</td><td>left or right IGBT, L=alternate gate type, D=parallel diode, dD=dotted connections</td></tr>
<tr><td>inductor(linespec,W,n,M)</td><td>cct</td><td>inductor, narrow or wide, 4 or n arcs, without or with magnetic core</td></tr>
<tr><td>integrator(linespec,size)</td><td>cct</td><td>integrating amplifier</td></tr>
<tr><td>intersect_(line1.start,line1.end,line2.start,line2.end)</td><td>gen</td><td>intersection of two lines</td></tr>
<tr><td>j_fet(linespec,L|R,P,E)</td><td>cct</td><td>left or right, N or P JFET, without or with envelope</td></tr>
<tr><td>larrow(label,->|<-,dist)</td><td>cct</td><td>arrow dist to left of last-drawn 2-terminal element</td></tr>
<tr><td>lbox(wid, ht, type)</td><td>gen</td><td>box oriented in current direction, type= e.g. dotted</td></tr>
<tr><td>left_</td><td>gen</td><td>left with respect to current direction</td></tr>
<tr><td>length3D(x,y,z)</td><td>3D</td><td>Euclidean length of triple x,y,z</td></tr>
<tr><td>lg_pin(location, logical name, pin label, n|e|s|w[N|L|M][E], pinno, optlen)</td><td>log</td><td>comprehensive logic pin; n|e|s|w=direction, N=negated, L=active low out, M=active low in, E=edge trigger</td></tr>
<tr><td>linethick_(number)</td><td>gen</td><td>set line thickness in points</td></tr>
<tr><td>lin_leng(line-reference)</td><td>gen</td><td>calculate the length of a line</td></tr>
<tr><td>ljust_</td><td>gen</td><td>ljust with respect to current direction</td></tr>
<tr><td>llabel(label,label,label)</td><td>cct</td><td>triple lable on left side of the element</td></tr>
<tr><td>loc_(x, y)</td><td>gen</td><td>location adjusted for current direction</td></tr>
<tr><td>log10E_</td><td>gen</td><td>constant $\log_{10}(e)$</td></tr>
<tr><td>loge</td><td>gen</td><td>logarithm, base e</td></tr>
<tr><td>lt_</td><td>gen</td><td>left with respect to current direction</td></tr>
<tr><td>manhattan</td><td>gen</td><td>sets direction cosines for left, right, up, down</td></tr>
<tr><td>mfpic_</td><td>mfpic</td><td>defined to signify mfpic is being used</td></tr>
<tr><td>microphone(U|D|L|R|degrees, size)</td><td>cct</td><td>microphone, In1 to In3 defined</td></tr>
<tr><td>mosfet(linespec,L|R,chars,E)</td><td>cct</td><td>MOSFET left or right, included components defined by characters, envelope</td></tr>
<tr><td>m4lstring(arg1,arg2)</td><td>gen</td><td>expand arg1 if it begins with sprintf or ", otherwise arg2</td></tr>
<tr><td>m4_arrow(linespec,ht,wid)</td><td>gen</td><td>arrow with adjustable head, filled when possible</td></tr>
<tr><td>m4xtract(‘string1’,string2)</td><td>gen</td><td>delete string2 from string1, return 1 if present</td></tr>
<tr><td>n_</td><td>gen</td><td>.n with respect to current direction</td></tr>
<tr><td>ne_</td><td>gen</td><td>.ne with respect to current direction</td></tr>
<tr><td>neg_</td><td>gen</td><td>unary negation</td></tr>
<tr><td>nport(box spec,nw,nn,ne,ns,space ratio,pin lgth,style)</td><td>cct</td><td>nport macro (default 2-port)</td></tr>
<tr><td>nw_</td><td>gen</td><td>.nw with respect to current direction</td></tr>
<tr><td>opamp(linespec,label,label,size,[P][R])</td><td>cct</td><td>operational amplifier with −, + or other internal labels, specified size. P adds power connections, R swaps In1, In2 labels</td></tr>
<tr><td>open_arrow(linespec,ht,wid)</td><td>gen</td><td>arrow with adjustable open head</td></tr>
</table>

par_(*element*,*element*,*separation*)	cct	two same-direction, same-length elements in parallel
point_(*angle*)	gen	(radians) set direction cosines
polar_(*x*,*y*)	gen	rectangular-to polar conversion
potentiometer(*linespec*,*cycles*,*fractional pos*,*length*,···)	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
psset_(*PSTricks settings*)	gen	set PSTricks parameters
pstricks_	pstricks	defined to signify PSTricks is being used
pt__	gen	TEX point-size factor, in scaled inches, (*scale/72.27)
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, drawn left or right of current direction
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
right_	gen	set current direction right
rjust_	gen	right justify with respect to current direction
rlabel(*label*,*label*,*label*)	cct	triple lable on right side of the element
rot3Dx(*radians*,x,y,z)	3D	rotates x,y,z about x axis
rot3Dy(*radians*,x,y,z)	3D	rotates x,y,z about y axis
rot3Dz(*radians*,x,y,z)	3D	rotates x,y,z about z axis
rpoint_(*linespec*)	gen	set direction cosines
rpos_(*position*)	gen	Here + *position*
rt_	gen	right with respect to current direction
rtod_	gen	constant, degrees/radian
rvec_(*x*,*y*)	gen	location relative to current direction
s_	gen	.s with respect to current direction
s_box(*text*,*expr1*,···)	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_wd(*name*,*default*)	gen	width of the most recent (or named) s_box
scr(*linespec*,R,G,E)	cct	triac (scr), right, gated, envelope
se_	gen	.se with respect to current direction
setview(*azimuth degrees*,*elevation degrees*)	3D	set projection viewpoint
sfg_init(*default line len, node rad, arrowhd len, arrowhd wid*)	cct	initialization of signal flow graph macros
sfgabove	cct	like above but with extra space
sfgbelow	cct	like below but with extra space
sfgarc(*linespec*,*text*,*text justification*,cw|ccw,*height scale factor*)	cct	directed arc drawn between nodes, with text label and a height-adjustment parameter
sfgline(*linespec*,*text*,*text justification*)	cct	directed straight line chopped by node radius, with text label
sfgnode(at *location*,*text*,above|below)	cct	white small circle, with text label
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*)		

<table>
<tr><td></td><td>gen</td><td>fill arbitrary closed curve</td></tr>
<tr><td>shadebox(*box specification*)</td><td>gen</td><td>box with edge shading</td></tr>
<tr><td>sign_(*number*)</td><td>gen</td><td>sign function</td></tr>
<tr><td>sind(*arg*)</td><td>gen</td><td>sine of an expression in degrees</td></tr>
<tr><td>source(*linespec*,V|v|I|i|AC|G|Q|X|*string*,*diameter*)</td><td>cct</td><td>source, blank or voltage (2 types) or current (2 types) or AC or G or Q or X or labelled</td></tr>
<tr><td>sourcerad_</td><td>cct</td><td>default source radius</td></tr>
<tr><td>sprod3D(a,x,y,z)</td><td>3D</td><td>scalar product of triple x,y,z by a</td></tr>
<tr><td>sp_</td><td>gen</td><td>evaluates to medium space for gpic strings</td></tr>
<tr><td>speaker(U|D|L|R|*degrees*,*size*,H)</td><td>cct</td><td>speaker, *In1* to *In7* defined; H=horn</td></tr>
<tr><td>sum_(*a*,*b*)</td><td>gen</td><td>binary sum</td></tr>
<tr><td>sum3D(x1,y1,z1,x2,y2,z2)</td><td>3D</td><td>sum of two triples</td></tr>
<tr><td>svec_(*x*,*y*)</td><td>log</td><td>scaled and rotated grid coordinate vector</td></tr>
<tr><td>sw_</td><td>gen</td><td>.sw with respect to current direction</td></tr>
<tr><td>switch(*linespec*,L|R,[C|O][D],B)</td><td>cct</td><td>SPST switch left or right, blank or closing or opening arrow, or button, D=dots</td></tr>
<tr><td>thicklines_(*number*)</td><td>gen</td><td>set line thickness in points</td></tr>
<tr><td>thinlines_(*number*)</td><td>gen</td><td>set line thickness in points</td></tr>
<tr><td>tline(*linespec*,*wid*,*ht*)</td><td>cct</td><td>transmission line, manhattan direction</td></tr>
<tr><td>transformer(*linespec*,L|R,np,A,ns)</td><td>cct</td><td>2-winding transformer: left or right, np primary arcs, air core, ns secondary arcs</td></tr>
<tr><td>twopi_</td><td>gen</td><td>2π</td></tr>
<tr><td>ujt(*linespec*,R,P,E)</td><td>cct</td><td>unijunction transistor, right, P-channel, envelope</td></tr>
<tr><td>unit3D(x,y,z)</td><td>3D</td><td>unit triple in the direction of triple x,y,z</td></tr>
<tr><td>up_</td><td>gen</td><td>set current direction up</td></tr>
<tr><td>up__</td><td>gen</td><td>up with respect to current direction</td></tr>
<tr><td>variable(‘*element*’,[A|P|L|[u]N][C|S],*angle*,*length*)</td><td>cct</td><td>overlaid arrow or line to indicate variable 2-terminal element: A=arrow, P=preset, L=linear, N=nonlinear, C=continuous, S=setpwise</td></tr>
<tr><td>vec_(*x*,*y*)</td><td>gen</td><td>position rotated with respect to current direction</td></tr>
<tr><td>vrot_(*x*,*y*,*xcosine*,*ycosine*)</td><td>gen</td><td>rotation operator</td></tr>
<tr><td>vlength(*x*,*y*)</td><td>gen</td><td>vector length $\sqrt{x^2+y^2}$</td></tr>
<tr><td>vscal_(*number*,*x*,*y*)</td><td>gen</td><td>vector scale operator</td></tr>
<tr><td>w_</td><td>gen</td><td>.w with respect to current direction</td></tr>
<tr><td>wid_</td><td>gen</td><td>width with respect to current direction</td></tr>
<tr><td>winding(L|R, *diam, pitch, turns, core wid, core color*)</td><td>cct</td><td>core winding drawn in the current direction; R=right-handed</td></tr>
<tr><td>xtal(*linespec*)</td><td>cct</td><td>quartz crystal</td></tr>
</table>

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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