M4 Macros for Electric Circuit Diagrams in LaTeX Documents

Dwight Aplevich

Contents, Version 11.2

1 Introduction 1

2 Using the macros 2
2.1 Quick start 2
2.1.1 Using m4 2
2.1.2 Processing with dpic and Tikz PGF or PSTricks . . . 3
2.1.3 Processing with gpic 4
2.1.4 Simplifications 4
2.2 Including the libraries 5

3 Pic essentials 6
3.1 Manuals 6
3.2 The linear objects: `line, arrow, spline, arc` 6
3.3 Positions, directions, rotations . . . 7
3.4 The planar objects: `box, circle, ellipse`, and text 8
3.5 Compound objects 8
3.6 Other language facilities 9

4 Two-terminal circuit elements 9
4.1 Circuit and element basics 9
4.2 The two-terminal elements 11
4.3 Branch-current arrows 16

5 Placing two-terminal elements 17
5.1 Labels 17
5.2 Series and parallel circuits 18

6 Composite circuit elements 20
6.1 Semiconductors 28

7 Corners 31

8 Looping 32

9 Logic gates 32
9.1 Automatic structures 36

10 Integrated circuits 38

11 Single-line diagrams 39
11.1 Two-terminal SLD elements 39
11.2 One-terminal and composite SLD elements 40

12 Element and diagram scaling 42
12.1 Circuit scaling 42
12.2 Pic scaling 42

13 Writing macros 43
13.1 Macro arguments 46

14 Interaction with LaTeX 47

15 PSTricks and other tricks 49
15.1 Tikz with pic 50

16 Web documents, `pdf`, and alternative output formats 51

17 Developer's notes 52

18 Bugs 53

19 List of macros 56

References 108

1 Introduction

> It appears that people who are unable to execute pretty pictures with pen and paper find it gratifying to try with a computer [11].

This manual[1] describes a method for drawing electric circuits and other diagrams in LaTeX and web documents. The diagrams are defined in the simple pic drawing language [9] augmented with m4 macros [10, 3], and are processed by m4 and a pic processor to convert them to Tikz PGF, PSTricks, other LaTeX-compatible code, SVG, or other formats. In its basic usage, the method has the advantages and disadvantages of TeX itself since it is macro-based and non-WYSIWYG, with ordinary text input. The book from which the above quotation is taken correctly points out

[1] This document is best displayed with a reader that shows bookmarks.

that the payoff can be in quality of diagrams at the price of the time spent in drawing them. The learning curve has been described [16] as "no worse than for a musical instrument." A quick reading of Section 2.1 will suffice for simple diagrams; virtuosity requires more practice. The basics of pic, which is Turing-complete, are *easy;* adding macros extends and specializes the language.

A collection of basic components, most based on IEC and IEEE standards [7, 8], and conventions for their internal structure are described. Macros such as these are only a starting point since it is often convenient to customize elements or to package combinations of them for particular drawings or contexts, processes for which m4 and pic are well suited.

2 Using the macros

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

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

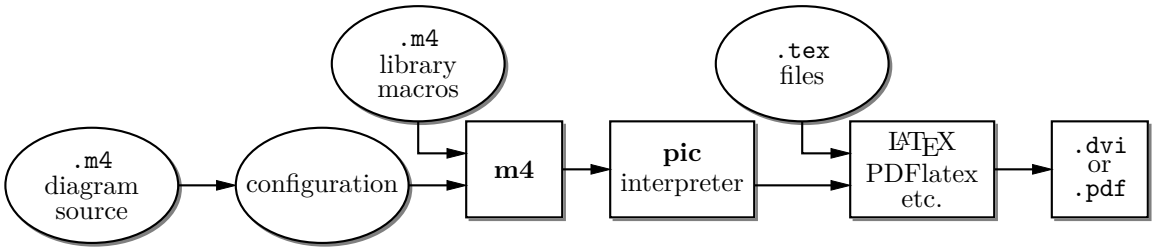


Figure 1: Inclusion of figures and macros in the LaTeX document.

The interpreter output contains Ti*k*z PGF [18] commands, PSTricks [20] commands, basic LaTeX graphics, tpic specials, or other formats, depending on the chosen options. These variations are described in Section 16.

There are two principal choices of pic interpreter. One is dpic, described later in this document. A partial alternative is GNU gpic -t (sometimes simply named pic) [12] together with a printer driver that understands tpic specials, typically dvips [15]. The dpic processor extends the pic language in small but important ways; consequently, some of the macros and examples in this distribution work fully only with dpic. Pic processors provide basic macro facilities, so some of the concepts applied here do not require m4.

2.1 Quick start

Read this section to understand basic usage of m4 and macros, and look at the `examples.pdf` file for cases that might be similar to yours. The contents of file `quick.m4` and resulting diagram are shown in Figure 2 to illustrate the language and the production of basic labeled circuits.

2.1.1 Using m4

The command

`m4` *filename* ...

causes m4 to search for the named files in the current directory and directories specified by environmental variable `M4PATH`. Set `M4PATH` to the full name (i.e., the path) of the directory containing `libcct.m4` and the other circuit library `.m4` files; otherwise invoke m4 as `m4 -I` *installdir* where *installdir* is the path to the directory containing the library files. Now there are at least two basic possibilities as follows, but be sure to read Section 2.1.4 for simplified use.

```
.PS                             # Pic input begins with .PS
cct_init                        # Read in macro definitions and set defaults
elen = 0.75                     # Variables are allowed; default units are inches
Vs: source(up_ elen); llabel(-,v_s,+)  # Name and label the source
   resistor(right_ elen); rlabel(,R,)  # Semicolon and line end are equivalent
   dot
   {                            # Save the current position and direction
      capacitor(down_ Vs.len); rlabel(+,v,-); llabel(,C,)
      dot
      }                         # Restore position and direction
   line right_ elen*2/3
   inductor(down_ Vs.len); rlabel(,L,); b_current(i)
   line to (Vs,Here)            # (Vs,Here) = (Vs.x,Here.y)
.PE                             # Pic input ends
```

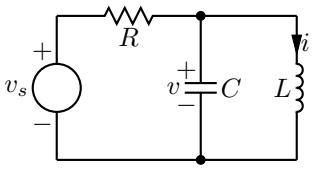


Figure 2: The file `quick.m4` and resulting diagram. There are several ways of drawing the same picture; for example, nodes can be defined (examples: `Origin: (0,0)` or `Northwest: Origin+(0,elen_)`) and circuit branches drawn between them; or absolute coordinates can be used (e.g., `source(up_ from (0,0) to (0,0.75))`). Element sizes can be varied and non-two-terminal elements included as described in later sections (Figure 25).

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

If you are using dpic with Ti*k*z PGF, put `\usepackage{tikz}` in the main LaTeX source file header and type the following commands or put them into a script or makefile:

```
m4 pgf.m4 quick.m4 > quick.pic
dpic -g quick.pic > quick.tex
```

To produce PSTricks code, the LaTeX header should contain `\usepackage{pstricks}`. The commands are modified to read `pstricks.m4` and invoke the `-p` option of dpic as follows:

```
m4 pstricks.m4 quick.m4 > quick.pic
dpic -p quick.pic > quick.tex
```

A configuration file (`pgf.m4` and `pstricks.m4` in the above examples) is *always* the first file to be given to m4. Put the following or its equivalent in the document body:

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

Then for Ti*k*z PGF, Invoking PDFlatex on the source produces `.pdf` output directly. For PSTricks, the commands "`latex` *file*`; dvips` *file*" produce *file*`.ps`, which can be printed or viewed using `gsview`, for example. The essential line is `\input quick` whether or not the `figure` environment is used.

The effect of the second m4 command above is shown in Figure 3.

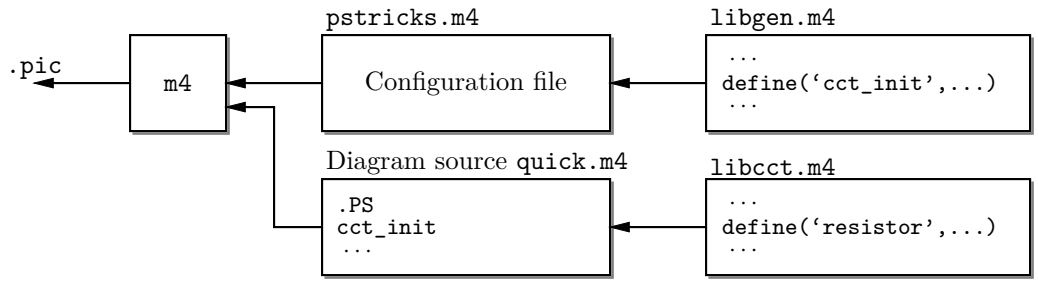


Figure 3: The command `m4 pstricks.m4 quick.m4 > quick.pic`.

Configuration files `pstricks.m4` and `pgf.m4` cause library `libgen.m4` to be read, thereby defining the macro `cct_init`. The diagram source file is then read and the circuit-element macros in `libcct.m4` are defined during expansion of `cct_init`.

2.1.3 Processing with gpic

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

```
m4 gpic.m4 quick.m4 > quick.pic
gpic -t quick.pic > quick.tex
```

and the figure inclusion statements are as shown:

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

2.1.4 Simplifications

M4 must read a configuration file before any other files, either before reading the diagram source file or at the beginning of it. There are several ways to control the process, as follows:

1. The macros can be processed by LaTeX-specific project software and by graphic applications such as Pycirkuit [13]. Alternatively when many files are to be processed, Unix "make," which is also available in PC and Mac versions, is a simple and powerful tool for automating the required commands. On systems without such facilities, a scripting language can be used.

2. The m4 commands illustrated above can be shortened to

   `m4 quick.m4 > quick.pic`

   by inserting `include(pstricks.m4)` (assuming PSTricks processing) *immediately* after the `.PS` line, the effect of which is shown in Figure 4. However, if you then want to use Ti*k*z PGF, the line must be changed to `include(pgf.m4)`.

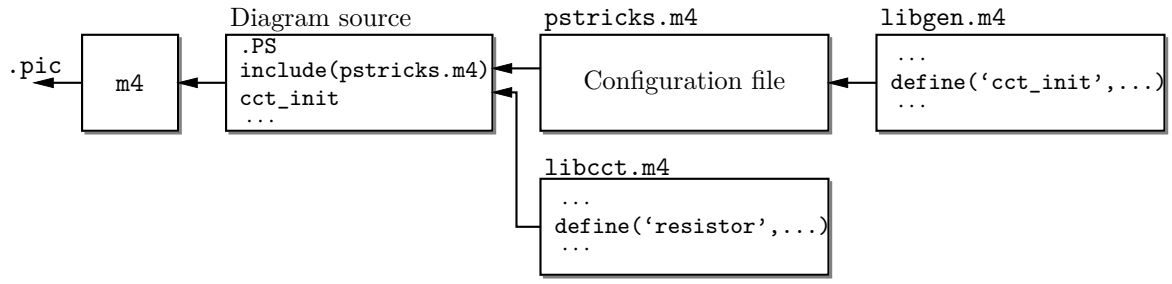


Figure 4: The command `m4 quick.m4 > quick.pic`, with `include(pstricks.m4)` preceding `cct_init`.

3. In the absence of a need to examine the file `quick.pic`, the commands for producing the `.tex` file can be reduced (provided the above inclusions have been made) to

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

4. You can put several diagrams into a single source file. Make each diagram 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
```
Produce a `.tex` file as usual, insert the `.tex` into the LaTeX source, and invoke the macros `\diaA` and `\diaB` at the appropriate places.

5. In some circumstances, it may be desirable to invoke m4 and dpic automatically from the document. Define a macro `\mtotex` as shown in the following example:

```
\documentclass{article}
\usepackage{tikz} % or \usepackage{pstricks}
\newcommand\mtotex[2]{\immediate\write18{m4 #2.m4 | dpic -#1 > #2.tex}}%
\begin{document}
\mtotex{g}{FileA} % Generate FileA.tex
\input{FileA.tex} \par
\mtotex{g}{FileB} % Generate FileB.tex
\input{FileB.tex}
\end{document}
```

   The first argument of `\mtotex` is a `p` for pstricks or `g` for pgf. Sources `FileA.m4` and `FileB.m4` must contain any required `include` statements, and the main document should be processed using the latex or pdflatex option `--shell-escape`. If the `M4PATH` environment variable is not set then insert `-I` *installdir* after `m4` in the command definition, where *installdir* is the absolute path to the installation directory. This method processes the picture source each time LaTeX is run, so for large documents containing many diagrams, the `\mtotex` lines could be commented out after debugging the corresponding graphic. A derivative of this method that allows the insertion of pic-produced code into a Ti*k*z picture is described in Section 15.1.

6. It might be convenient for the source of small diagrams to be part of the document source text. The `filecontents` environment of current LaTeX allows this; older versions can employ a now-obsolete package `filecontents.sty`. The following example for processing by `pdflatex --shell-escape` first writes the m4 source to file `sample.m4`, invokes `\mtotex` on it, and reads in the result:

```
\begin{filecontents}[overwrite,noheader,nosearch]{sample.m4}
include(pgf.m4)
.PS
cct_init
```
drawing commands ...
```
.PE
\end{filecontents}
\mtotex{g}{sample}
\input{sample.tex}
```

2.2 Including the libraries

The configuration files for dpic are as follows, depending on the output format (see Section 16): `pstricks.m4`, `pgf.m4`, `mfpic.m4`, `mpost.m4`, `postscript.m4`, `psfrag.m4`, `svg.m4`, `gpic.m4`, or `xfig.m4`. The usual case for producing circuit diagrams is to read `pgf.m4` or `pstricks.m4` first when dpic is the postprocessor or to set one of these as the default configuration file. For gpic, the configuration file is `gpic.m4`.

At the top of each diagram source, put one or more initialization commands; that is, `cct_init, log_init, sfg_init, darrow_init, threeD_init,` or, for diagrams not requiring specialized macros, `gen_init`. As shown in Figures 3 and 4, each initialization command reads in the appropriate macro library if it hasn't already been read; for example, `cct_init` tests whether `libcct.m4` has been read and includes it if necessary.

The distribution includes a collection of pic utilities in the file `dpictools.pic,` which is loaded automatically by macros that invoke the `NeedDpicTools` macro.

The file libSLD.m4 contains macros for drawing single-line power distribution diagrams. The line `include(libSLD.m4)` loads the macros. A few of the distributed example files contain other macros that can be pasted into diagram source files; see `Flow.m4` or `Buttons.m4`, for example.

Also included in the distribution is a generous set of examples to show capabilities of the macros and to act as a source of code if you wish to produce similar diagrams.

The libraries contain hints and explanations that might help in debugging or if you wish to modify any of the macros. Macros are generally named using the obvious circuit element names so that programming becomes something of an extension of the pic language. Some macro names end in an underscore to reduce the chance of name clashes. These can be invoked in the diagram source but there is no long-term guarantee that their names and functionality will remain unchanged. Finally, macros intended only for internal use begin with the characters `m4`.

3 Pic essentials

Pic source is a sequence of lines in a text 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 compound object, or `block,` is planar and consists of a pair of square brackets enclosing other objects, as described in Section 3.5.

Objects can be placed using absolute coordinates or, as is often better, 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 letter.

3.1 Manuals

The classic pic manual [9] is still a good introduction to pic, but a more complete manual [14] can be found in the GNU groff package, and both are available on the web [9, 14]. Reading either will give you basic competence with pic in an hour. Explicit mention of `*roff` string and font constructs in these manuals should be replaced by their equivalents in the LaTeX context. The dpic manual [1] includes a man-page lanuage summary in an appendix.

A web search will yield good discussions of "little languages"; for pic in particular, see Chapter 9 of [2]. Chapter 1 of reference [5] also contains a brief discussion of this and other languages.

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

A line can be drawn as follows:

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

where *position* is defined below or

`line` *direction distance*

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

```
scale = 25.4
```

has the effect of changing the units to millimetres, as described in Section 12.

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

```
line up_ 3/sqrt(2) right_ 3/sqrt(2) dashed
```
draws a line 3 units long from the current location, at a 45° angle above horizontal. Lines (and other objects) can be specified as `dotted,` `dashed,` or `invisible,` as above.

The construction

```
line from A to B chop x
```
truncates the line at each end by `x` (which may be negative) or, if `x` is omitted, by the current circle radius, a convenience when A and B are circular graph nodes, for example. Otherwise

```
line from A to B chop x chop y
```
truncates the line by `x` at the start and `y` at the end.

Any of the above means of specifying line direction and length will be called a *linespec*.

Lines can be concatenated to create multsegmented objects. For example, to draw a triangle:

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

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.

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, and that drawing an arc may change the current drawing direction. This distribution includes the m4 macros

`arcr(` *position, radius, start radians, end radians, modifiers, ht*`)`

`arcd(` *position, radius, start degrees, end degrees, modifiers, ht*`)`

`arca(` *chord linespec,* `ccw|cw,` *radius, modifiers*`)`

to draw uniquely defined arcs. If the fifth argument of `arcr` or `arcd` contains `->` or `<-` then a midpoint arrowhead of height specified by arg6 is added. For example,

```
arcd((1,-1),,0,-90,<- outlined "red") dotted
```
draws a red dotted arc with midpoint arrowhead, centre at $(1, -1)$, and default radius. The example

```
arca(from (1,1) to (2,2),,1,->)
```
draws an acute angled arc with arrowhead on the chord defined by the first argument.

3.3 Positions, directions, rotations

A *position* can be defined by a coordinate pair; e.g., `3,2.5`, more generally using parentheses by `(`*expression, expression*`)`, as a sum or difference; e.g., *position* + `(`*expression, expression*`)`, or by the construction `(`*position, position*`)`, the latter taking the x-coordinate from the first position and the y-coordinate from the second. A position can be given a symbolic name beginning with an upper-case letter, e.g. `Top: (0.5,4.5)`. Such a definition does not affect the calculated figure boundaries. The current position `Here` is always defined and is equal to $(0, 0)$ at the beginning of a diagram or block. The coordinates of a position are accessible, e.g. `Top.x` and `Top.y` can be used in expressions. The center, start, and end of linear objects (and the defined points of other objects as described below) are predefined 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
```

Pic stores the current drawing direction, which is unfortunately limited to `up, down, left, right,` for use when necessary. The circuit 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, a transformation matrix is applied at the macro level to generate the required (but sometimes very elaborate) pic code. The key variable is `rp_ang`, the drawing direction in radians, with 0 to the right. Potentially, the matrix elements can be used for other transformations. The macro

`setdir_(`*direction, default direction*`)`

is preferred when setting drawing direction. The *direction* arguments are of the form

`R[ight] | L[eft] | U[p] | D[own] |` *degrees*,

but the macros `Point_(`*degrees*`)`, `point_(`*radians*`)`, and `rpoint_(`*relative linespec*`)` are employed in many macros to re-define the entries of the matrix (named `m4a_`, `m4b_`, `m4c_`, and `m4d_`) for the required rotation. The macro `eleminit_` in the two-terminal elements invokes `rpoint_` with a specified or default *linespec* to establish element length and direction.

The dpic primitive `arc` and some of the circuit element macros explicitly or implicitly change the drawing direction so it may be necessary to set it explicitly for succeeding elements.

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

3.4 The planar objects: `box`, `circle`, `ellipse`, and text

Planar objects are drawn by specifying the width, height, and 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`. The compass points `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`. Planar objects can also be placed by specifying the location of a defined point; 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. For example, either

`box dashed fill_(`*expression*`)` or `box dashed outlined "`*color*`" shaded "`*color*`"`

produces a dashed box. The first case has a gray fill determined by *expression*, with 0 corresponding to black and 1 to white; the second case allows color outline and fill, the color strings depending on the postprocessor. Postprocessor-compatible RGB color strings are produced by the macro `rgbstring(`*red fraction, green fraction, blue fraction*`)`; to produce an orange fill for example:

```
... shaded rgbstring( 1, 0.645, 0)
```

Basic colours for lines and fills are provided by gpic and dpic, but more elaborate line and fill styles or other effects can be incorporated, depending on the postprocessor, using

`command "`*string*`"`

where *string* is one or more postprocessor command lines.

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 14. The second occurrence associates one or more strings with an object, e.g., the following writes two words, one above the other, at the centre of an ellipse:

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

The C-like pic function `sprintf("`*format string*`",`*numerical arguments*`)` is equivalent to a string. (Its implementation passes arguments singly to the C `snprintf` function).

3.5 Compound objects

A compound object is a group of statements enclosed in square brackets. Such an object, often called a *block*, is placed by default as if it were a box regardless of its contents, but it can also be

placed by specifying the final position of a defined point. A defined point is the center or compass corner of the bounding box of the object, an interior location, or a defined point of one of its internal objects. Consider the last line of the code fragment shown:

```
Ands: [ And1: AND_gate
          And2: AND_gate at And1 - (0,And1.ht*3/2)
        ] with .And2.In1 at position
```

The two gate macros evaluate to compound objects containing `Out`, `In1`, and other locations. The final positions of all objects inside the square brackets are determined in the last line by specifying the position of `In1` of gate `And2`. The compound block has been given the name `Ands`.

3.6 Other language facilities

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

Another possibility for defining positions is

expression `between` *position* `and` *position*

which means the vector expression

1st position + (*2nd position* − *1st position*) × *expression*

and which can be abbreviated as

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

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

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

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

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

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

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

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

the looping facility

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

operating-system commands, pic macros, and external file inclusion.

4 Two-terminal circuit elements

There is a fundamental difference between the two-terminal elements, each of which is drawn along an invisible straight-line segment, and other elements, which are generally compound objects in `[ ]` blocks as described in Section 3.5 and Section 6. Anticipating later discussion for example, `resistor(from A to B)` draws a (two-terminal) resistor from A to B, whereas `transformer with .P1 at C` draws a (non-two-terminal) transformer with internal point `P1` at location `C`.

The two-terminal element macros follow a set of conventions described in this section, and other elements will be described in Section 6.

4.1 Circuit and element basics

A list of the library macros and their arguments is in Section 19. The arguments have default values, so that only those that differ from defaults need be specified.

Figure 5 shows a resistor and serves as an example of pic commands. The first part of the source file for this figure is on the left:

```
.PS
  cct_init
  linewid = 2.0
  linethick_(2.0)
R1:  resistor
```

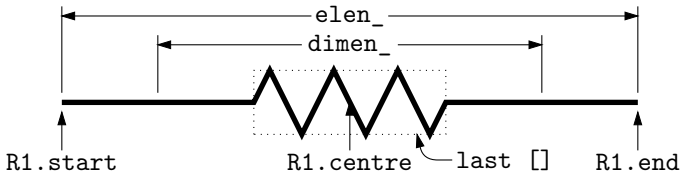


Figure 5: Resistor named `R1`, showing the size parameters, enclosing block, and predefined positions.

The lines of Figure 5 and the remaining source lines of the file are explained below:

- The first line after `.PS` invokes the macro `cct_init` that loads the library `libcct.m4` and initializes local variables needed by circuit-element macros.
- The sizes of circuit elements are proportional to the pic environmental variable `linewid`, so redefining this variable changes element sizes. The element body is drawn in proportion to `dimen_`, a macro that evaluates to `linewid` unless redefined, and the default element length is `elen_`, which evaluates to `dimen_*3/2` unless redefined. Setting `linewid` to 2.0 as in the example means that the default element length becomes $2.0 \times 3/2 = 3.0$ in. For resistors, the default length of the body is `dimen_/2`, and the width is `dimen_/6`. All of these values can be customized. Element scaling and the use of SI units is discussed further in Section 12.
- The macro `linethick_` sets the default thickness of subsequent lines (to 2.0 pt in the example). 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 macro arguments because m4 and dpic ignore white space immediately preceding arguments. Otherwise, a long line can be continued to the next by putting a backslash as the rightmost character.
- The two-terminal element macros expand to sequences of drawing commands that begin with '`line invis` *linespec*', where *linespec* is the first argument of the macro if it is non-blank, otherwise the line is drawn a distance `elen_` in the current direction, which is to the right by default. The invisible line is first drawn, then the element is drawn on top of it. The element—rather, the initial invisible line—can be given a name, `R1` in the example, so that positions `R1.start`, `R1.centre`, and `R1.end` are automatically defined as shown.
- The element body is drawn in or overlaid by a block, which can be used to place labels around the body. The block corresponds to an invisible rectangle with horizontal top and bottom lines, regardless of the direction in which the element is drawn. A dotted box has been drawn in the diagram to show the block boundaries.
- The last sub-element, identical to the first in two-terminal elements, is an invisible line that can be referenced later to place labels or other elements. If you create your own macros, you might choose simplicity over generality, and include only visible lines.

To produce Figure 5, the following embellishments were added after the previously shown source:

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

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

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

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

- The line thickness is set to the default thin value of 0.4 pt, and the box displaying the element body block is drawn. Notice how the width and height can be specified, and the box centre positioned at the centre of the block.
- The next paragraph draws two objects, a spline with an arrowhead, and a string left-justified at the end of the spline. Other string-positioning modifiers than `ljust` are `rjust, above,` and `below.`
- The last paragraph invokes a macro for dimensioning diagrams.

4.2 The two-terminal elements

Two-terminal elements are shown in Figures 6 to 15 and part of Figure 16. Several are included more than once to illustrate some of their arguments, which are listed in detail in Section 19.

Most of the two-terminal elements are oriented; that is, they have a defined direction or polarity. Several element macros include an argument that reverses polarity, but there is also a more general mechanism, as follows. The first argument of the macro

`reversed(‘`*macro name*`’,`*macro arguments*`)`

is the name of a two-terminal element in quotes, followed by the element arguments. The element is drawn with reversed direction; thus,

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

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

Similarly, the macro

`resized(`*factor*`,‘`*macro name*`’,`*macro arguments*`)`

will resize the body of an element by temporarily multiplying the `dimen_` macro by *factor* but m4 primitives can be employed instead as follows:

`pushdef(‘dimen_’,dimen_*(`*factor*`)),`*macro name*`(`*arguments*`) popdef(‘dimen_’)`

More general resizing should be done by redefining `dimen_` globally as described in Section 12.1.

Figure 6 shows some resistors with typical variants. The first macro argument specifies the

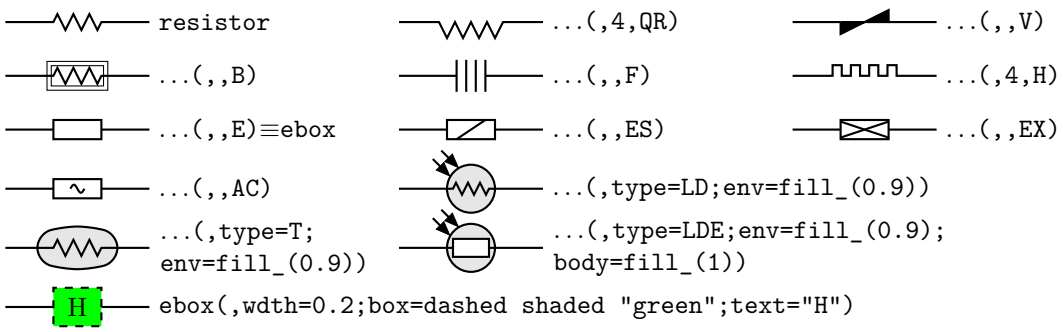


Figure 6: Resistors drawn by the macro `resistor(`*linespec, n, chars, cycle wid*`)`. The second argument is either an integer to specify number of cycles or blank for the default (3). The third argument specifies the desired variant with `R` added for orientation to the right. The default `ebox` element designates a box resistor. The alternative invocation is `resistor(`*linespec, key=value sequence*`)` illustrated in the three bottom rows.

invisible line segment along which the element is drawn. If the argument is blank, the element is drawn from the current position in the current drawing direction along a default length. 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.

Capacitors are illustrated in Figure 7. See Section 6 for the `variable` macro.

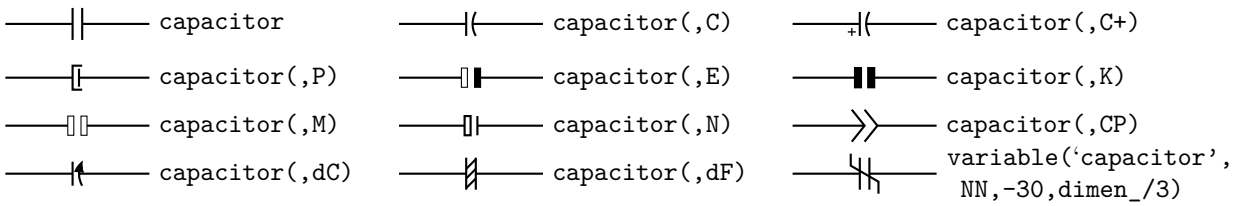


Figure 7: The `capacitor(`*linespec*, *chars*, `[R],`*height*, *width*`)` macro, and an example application of the `variable` macro.

Inductors are illustrated in Figure 8.

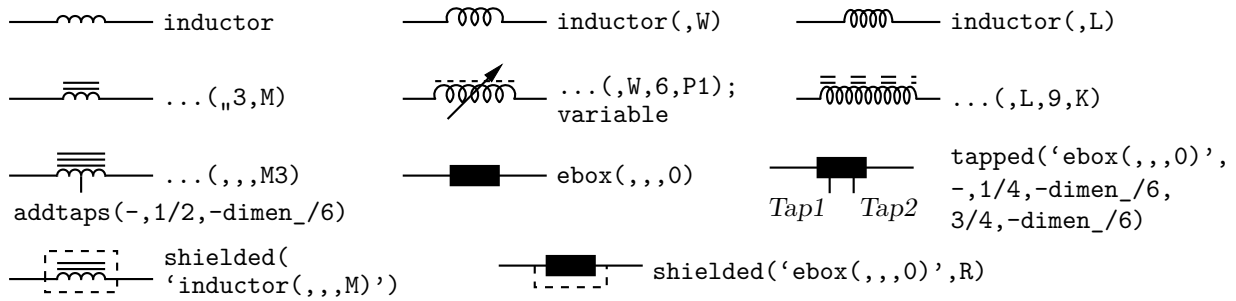


Figure 8: Basic inductors created with the `inductor(`*linespec*, `W|L,` *cycles*, `M|P|K,` *loop wid*`)` macro, the `ebox` macro for European-style inductors, and some modifications (see also Section 6). When an embellished element is repeated several times, writing a wrapper macro may be desirable.

Some two-terminal elements often drawn with truncated leads are in Figure 9. More basic elements are in Figure 10, and amplifiers in Figure 11.

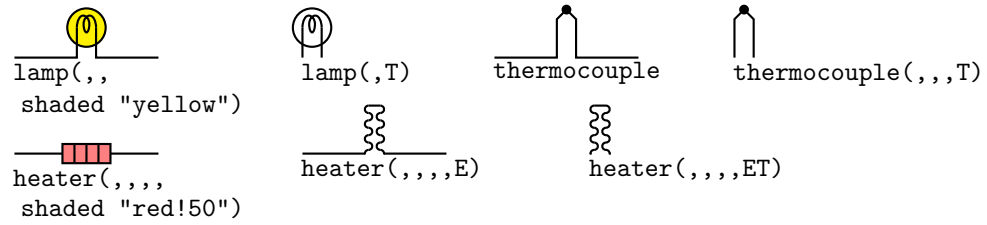


Figure 9: These elements have two terminals but are often drawn with truncated leads.

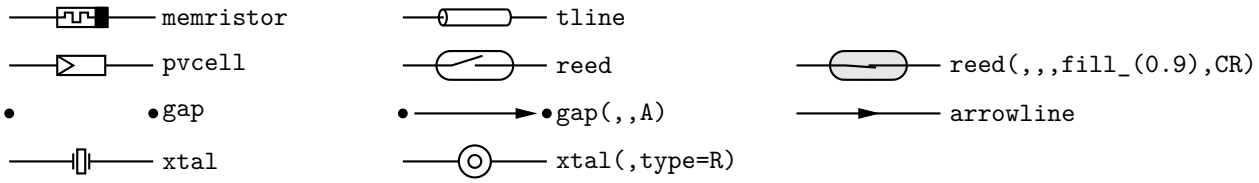


Figure 10: More two-terminal elements.

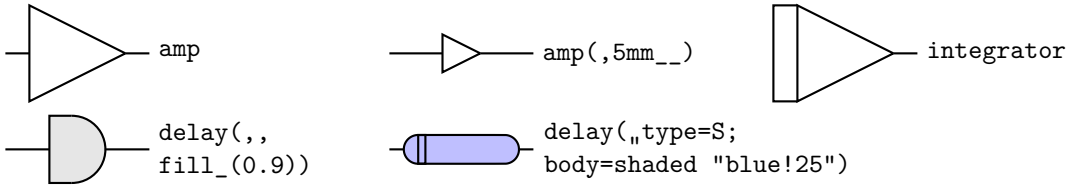


Figure 11: Amplifier, delay, and integrator.

Diodes are shown in Figure 12.

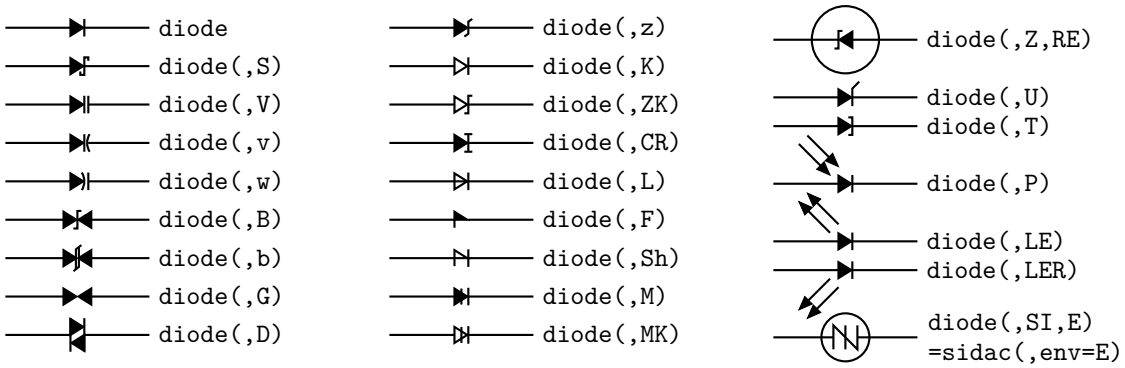


Figure 12: The macro `diode(`*linespec*`, B|b|CR|D|F|G|L|LE[R]|P[R]|S|Sh|SI|T|U|V|v|w|Z|z|`*chars*`, [R][E])`. Appending `K` to the second argument draws an open arrowhead. The option `diode(,SI)` is a wrapper for the `sidac(`*linespec, keys*`)` macro.

The arrows are drawn relative to the diode direction by the `LE` option. For absolute arrow directions, one can define a wrapper (see Section 13) for the `diode` macro to draw arrows at 45 degrees, for example:

```
define('myLED','diode('$1'); em_arrows(N,45) with .Tail at last [].ne')
```

Figure 13 shows sources, many of which contain internal symbols, and of which the `AC` and `S` options illustrate the need to draw a single cycle of a sinusoid or approximate sinusoid.

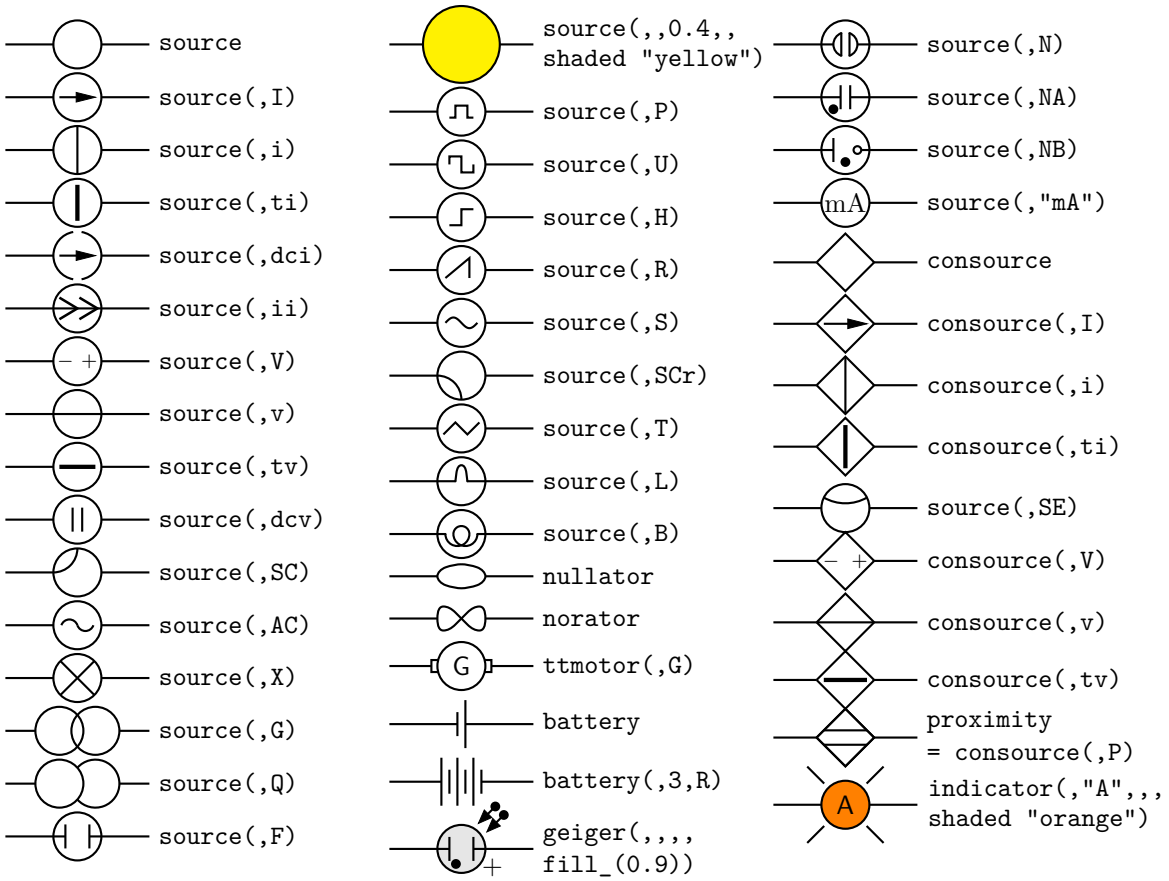


Figure 13: Sources and source-like elements. An argument of each element allows customization such as shading. The `geiger` macro is a wrapper for `source`.

As a convenience, the macro `ACsymbol(at` *position, length, height,* `[n:][A]U|D|L|R|`*degrees*) defines an interface to the `sinusoid` macro. For example, to add the symbol "∼" to an ebox:

```
ebox; { ACsymbol(at last [],,,dimen_/8) }
```

For direct current (⎓), there is also `DCsymbol(at` *position, length, height,* `U|D|L|R|`*degrees*), and for power-system diagrams, macros `Deltasymbol(at` *position, keys,* `U|D|L|R|`*degrees*), and `Ysymbol(at` *position, keys,* `U|D|L|R|`*degrees*) that generate symbols "Δ" and "Y" with possible options respectively.

Fuses, breakers, and jumpers are in Figure 14, and switches with numerous controls in Figure 15.

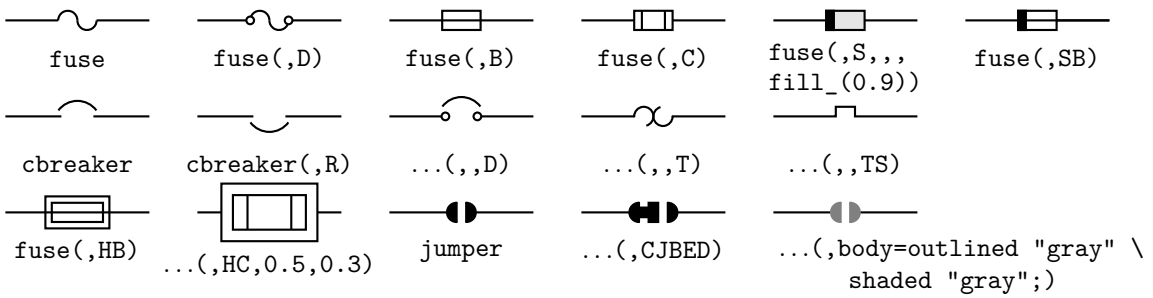


Figure 14: Variations of the macros `fuse(`*linespec*`, A|dA|B|C|D|E|S|HB|HC|SB,` *wid*`,` *ht*`,` *attributes*`)`, `cbreaker(`*linespec*`,L|R,D|T|TS)`, and `jumper(`*linespec*`,`*chars*`|`*keys*`)`.

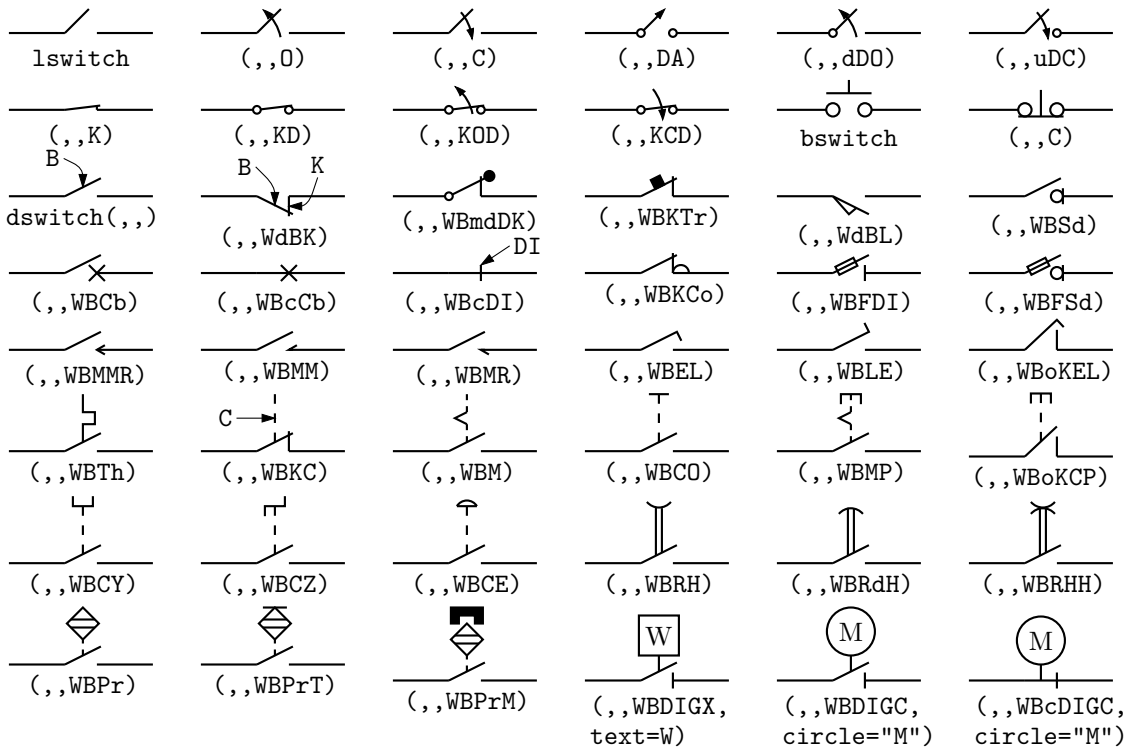


Figure 15: The `switch(`*linespec*`,L|R,`*chars*`,L|B|D,`*attribs*`)` macro is a wrapper for the macros `lswitch(`*linespec*`,[L|R],[O|C][D][K][A])`, `bswitch(`*linespec*`,[L|R],[O|C])`, and the many-optioned `dswitch(`*linespec*`,R,W[ud]B` *chars,attributes*`)` shown. The switch is drawn in the current drawing direction. A second-argument `R` produces a mirror image with respect to the drawing direction. The separately defined macros `Proxim` and `Magn` embellish switches in the bottom row.

Figure 16 shows a collection of surge-protection devices, or arresters, of which the `E` and `S` types may be either 2-terminal or as 3-terminal (composite) elements described in Section 6.

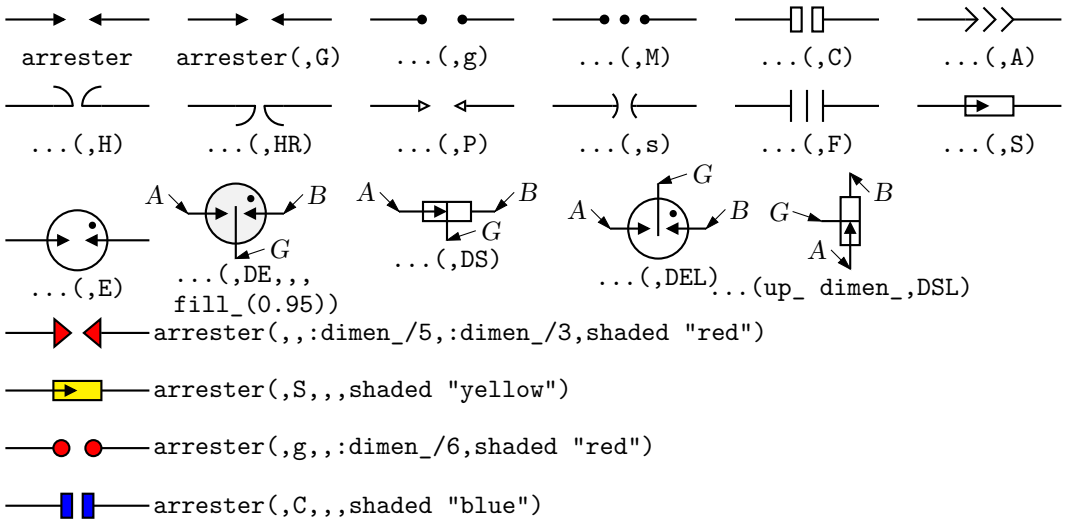


Figure 16: Variations of the macro `arrester(`*linespec, chars, body len*`[:`*arrowhead ht*`], `*body ht*`[:`*arrowhead wid*`], `*attributes*`)`, with some examples below. Putting `D` in argument 2 for the `S` or `E` configuration creates a 3-terminal composite element with terminals *A*, *B*, and *G*, in which case the first argument determines length and direction but not position.

Elements for use in ladder diagrams are in Figure 17.

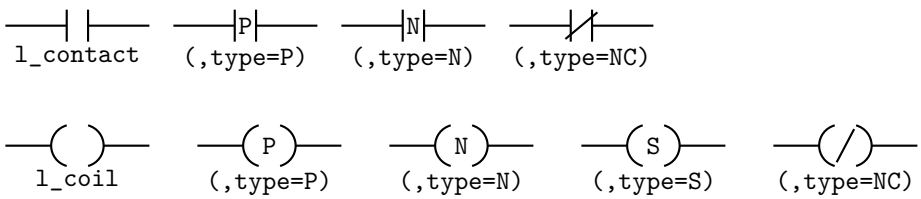


Figure 17: The `contact(`*linespec, keys*`)` and `coil(`*linespec,keys*`)` macros for ladder diagrams.

Figure 18 contains radiation-effect arrows for embellishing two-terminal and other macros.

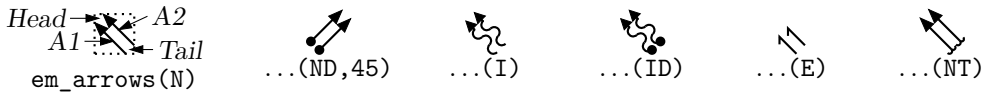


Figure 18: Radiation arrows: `em_arrows(`*type*`|`*keys,angle, length*`)`

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 19 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, NN` 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 5.1 by specifying an empty first argument; thus, the following line draws the third resistor in Figure 19:

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

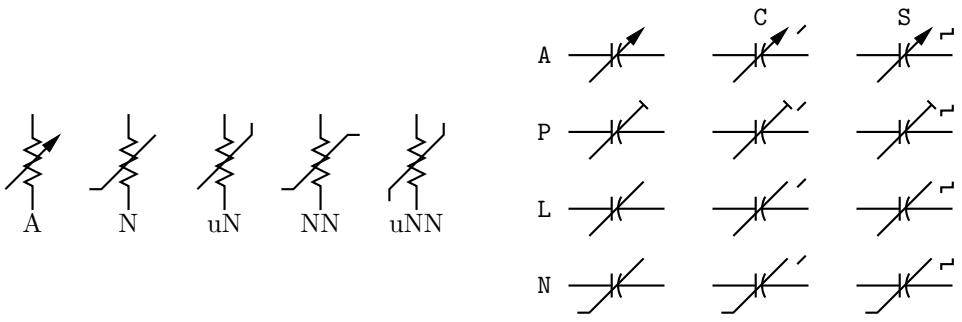


Figure 19: Illustrating `variable('element',[A|P|L|[u]N]|[u]NN]][C|S],[+|-]angle,length)`. For example, `variable('resistor(up_ dimen_)',A)` draws the leftmost resistor shown above. The default angle is 45°, regardless of the direction of the element, but the angle preceded by a sign (+ or −) is taken to be relative to the drawing direction of the element as for the lower right capacitor in Figure 7, for example. The array on the right shows the effect of the second argument.

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 labelling two-terminal elements, as shown in Figure 20.

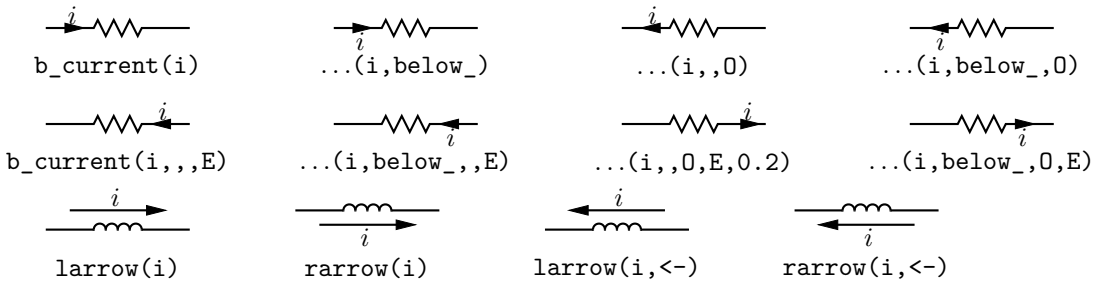


Figure 20: Illustrating `b_current, larrow,` and `rarrow`. The drawing direction is to the right.

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.

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 20. 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.

5 Placing two-terminal elements

The length and position of a two-terminal element are defined by a straight-line segment, so four numbers or equivalent are required to place the element as in the following example:

`resistor(from (1,1) to (2,1)).`

However, pic has a very useful concept of the current point (named `Here`); thus,

`resistor(to (2,1))`

is equivalent to

`resistor(from Here to (2,1)).`

Any defined position can be used; for example, if *C1* and *L2* are names of previously defined two-terminal elements, then, for example, the following places the resistor:

`resistor(from L2.end to C1.start)`

A line segment starting at the current position can also be defined using a direction and length. To draw a resistor up d units from the current position, for example:

`resistor(up_ d)`

5.1 Labels

Arbitrary text labels can be positioned by any pic placement method including the basic examples shown:

"*text*" `at` *position*
"*text*" `at` *position* `above`
"*text*" `wid` *width* `ht` *height* `with .sw at` *position*

In addition, special macros for labeling two-terminal elements are available:

`llabel(` *label, label, label, rel placement, block name* `)`
`clabel(` *label, label, label, rel placement, block name* `)`
`rlabel(` *label, label, label, rel placement, block name* `)`
`dlabel(` *long, lat, label, label, label,* `[X][A|B][L|R])`

The first macro places the first three arguments, which are treated as math-mode strings, on the left side of the last `[]` block (or the block named in the fifth argument if present) *with respect to the current direction:* `up, down, left, right.` The second macro places the strings along the centre of the element, and the third along the right side. The fourth applies a displacement (*long, lat*) with respect to the drawing direction. Labels beginning with `sprintf` or a double quote are copied literally rather than assumed to be in math mode. A simple circuit example with labels is shown in Figure 21.

```
.PS
# 'Loop.m4'
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
```

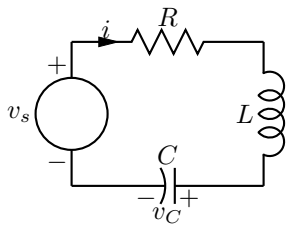


Figure 21: A loop containing labeled elements, with its source code.

Most often, only the first three label arguments are needed, and blank arguments are ignored. The fourth argument can be `above, below, left,` or `right` to supplement the default relative position. 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. In the fourth argument, an `X` aligns the labels with respect to the line joining the two terminals rather than the element body, and `A, B, L, R` use absolute `above, below, left,` or `right` alignment respectively for the labels.

As shown in Figure 22, "`Point_(-30); resistor`" draws a resistor along a line with slope of -30 degrees; similarly, "`rpoint_(to Z)`" sets the current direction cosines to point from the current location to location Z.

```
.PS
# 'Oblique.m4'
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,-0.1,,L_3,)
R:dot
L:dot( at Cl-(R.x-Cr.x,Cr.y-R.y) )

   inductor(from L to Cl); dlabel(0,-0.12,,L_2,)
   right_; resistor(from L to R); rlabel(,R_2,)
   resistor(from T to R); dlabel(0,0.15,,R_3,) ; b_current(\;y,ljust)
   line from L to 0.2<L,T>
   source(to 0.5 between L and T); dlabel(sourcerad_+0.07,0.1,-,,+)
      dlabel(0,sourcerad_+0.07,,u,)
   resistor(to 0.8 between L and T); dlabel(0,0.15,,R_1,)
   line to T
.PE
```

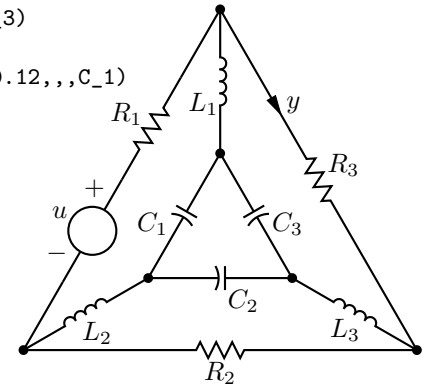


Figure 22: Illustrating elements drawn at oblique angles.

Figure 22 illustrates that some hand placement of labels using `dlabel` may be useful when elements are drawn obliquely. The figure also illustrates that any commas within m4 arguments must be treated specially because the arguments are separated by commas. Argument commas are protected either by parentheses as in `inductor(from Cr to Cr+vec_(elen_,0))`, or by multiple single quotes as in `‘‘,’’`, as necessary. Commas also may be avoided by writing `0.5 between L and T` instead of `0.5<L,T>`.

5.2 Series and parallel circuits

To draw elements in series, each element can be placed by specifying its line segment as described previously, but the pic language makes some geometries particularly simple. Thus,

```
setdir_(Right)
resistor; llabel(,R); capacitor; llabel(,C); inductor; llabel(,L)
```

draws three elements in series as shown in the top line of Figure 23.

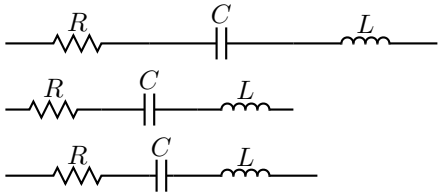


Figure 23: Three ways of drawing basic elements in series.

However, the default length `elen_` appears too long for some diagrams. It can be redefined temporarily (to `dimen_`, say), by enclosing the above line in the pair

```
pushdef('elen_',dimen_) resistor... popdef('elen_')
```

with the result shown in the middle row of the figure.

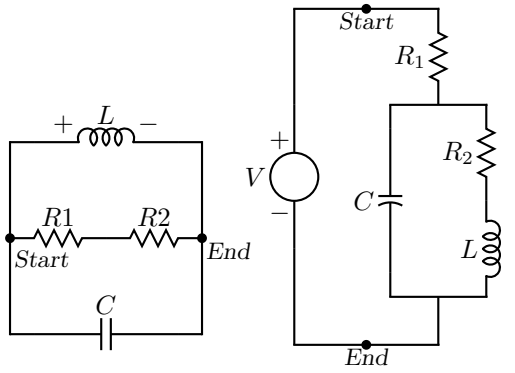
Alternatively, the length of each element can be tuned individually; for example, the capacitor in the above example can be shortened as shown, producing the bottom line of Figure 23:

```
resistor; llabel(,R)
capacitor(right_ dimen_/4); llabel(,C)
inductor; llabel(,L)
```

If a macro that takes care of common cases automatically is to be preferred, you can use the macro `series_(`*elementspec, elementspec, ...*`)`. This macro draws elements of length `dimen_` from the current position in the current drawing direction, enclosed in a `[ ]` block. The internal names `Start`, `End`, and `C` (for centre) are defined, along with any element labels. An *elementspec* is of the form `[`*Label*`:]` *element*`;` `[`*attributes*`]`, where an attribute is zero or more of `llabel(...)`, `rlabel(...)`, or `b_current(...)`.

Drawing elements in parallel requires a little more effort but, for example, three elements can be drawn in parallel using the code snippet shown, producing the left circuit in Figure 24:

```
define(‘elen_’,dimen_)
L: inductor(right_ 2*elen_,W); llabel(+,L,-)
R1: resistor(right elen_ from L.start+(0,-dimen_)); llabel(,R1)
R2: resistor; llabel(,R2)
C: capacitor(right 2*elen_ from R1.start+(0,-dimen_)); llabel(,C)
   line from L.start to C.start
   line from L.end to C.end
```




```
setdir_(Down)
parallel_(
 series_(‘R1:resistor; rlabel(,R_1)’,
  parallel_(
   series_(‘resistor; rlabel(,R_2)’,
           ‘inductor(,W); rlabel(,L)’),
   ‘capacitor(,C); rlabel(,C)’ ),
  line down dimen_/2),
 ‘Sep=linewid*3/2; V:source; rlabel(+,V,-)’)
```

```
parallel_( ‘L:inductor(,W); llabel(+,L,-)’,
 series_(‘R1:resistor; llabel(,R1)’, ‘R2:resistor; llabel(,R2)’),
 ‘C:capacitor; llabel(,C)’ )
```

Figure 24: Illustrating the macros `parallel_` and `series_`, with `Start` and `End` points marked.

A macro that produces the same effect automatically is

`parallel_(‘`*elementspec*`’, ‘`*elementspec*`’, ...)`

The arguments *must be quoted* to delay expansion, unless an argument is a nested `parallel_` or `series_` macro, in which case it is not quoted. The elements are drawn in a `[ ]` block with defined points `Start`, `End`, and `C`. An *elementspec* is of the form

`[Sep=`*val*`;][`*Label*`:]` *element*`;` `[`*attributes*`]`

where an *attribute* is of the form

`[llabel(...);] | [rlabel(...)] | [b_current(...);]`

Putting `Sep=`*val*`;` in the first branch sets the default separation of all branches to *val*; in a later element, `Sep=`*val*`;` applies only to that branch. An element may have normal arguments but should not change the drawing direction.

6 Composite circuit elements

Many basic elements are not two-terminal. These elements are usually enclosed in a `[ ]` pic block, and contain named interior locations and components. Nearly all elements drawn within blocks can be customized by adding an extra argument, which is executed as the last item within the block. By default, a block is placed as if it were a box; otherwise, the block must be placed by using its compass corners, thus: *element* `with` *corner* `at` *position* or, when the block contains predefined locations, thus: *element* `with` *location* `at` *position*. In some cases, an invisible line can be specified by the first argument to determine length and direction (but not position) of the block. A few macros are positioned with the first argument; the `ground` macro, for example: `ground(at` *position*`)`.

Figure 25 illustrates the adaptation of file `quick.m4` to include a transformer, a composite element described in detail below, followed by code for the figure.

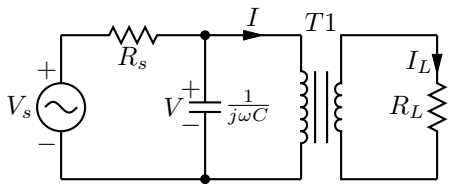


Figure 25: The file `quick.m4` modified to include a transformer, a composite element positioned by placing an internal point: `T1: transformer(down_ Vs.len,,6,,4) with .P1 at Here`.

Figure 26 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* but not the position of the transformer, which is determined by the enclosing block as normal for a composite element. The second argument places the secondary side of the transformer to the left or right of the drawing direction. The optional third and fifth arguments specify the number of primary and secondary arcs respectively. If the fourth argument string contains an `A`, the iron core is omitted; if a `P`, the core is dashed (powder); and if it contains a `W`, wide windings are drawn. A `D1` puts phase dots at the *P1, S1* end, `D2` at the *P2, S2* ends, and `D12` or `D21` puts dots at opposite ends.

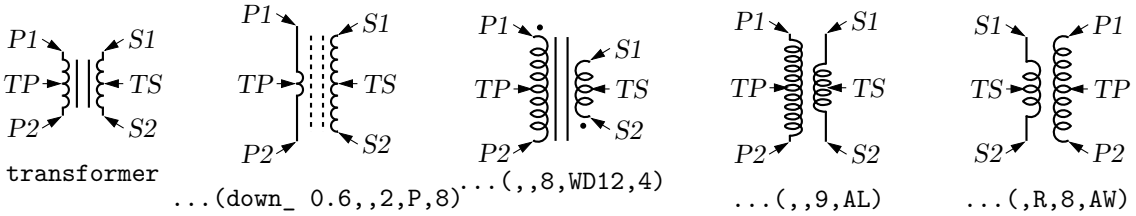


Figure 26: The `transformer(`*linespec*`,L|R,`*np*`,[A|P][W|L][D1|D2|D12|D21],`*ns*`)` macro (drawing direction `down`), showing predefined terminal and centre-tap points.

The code for Figure 25 is reproduced in the following. The transformer is positioned by placing internal point *P1*.

```
.PS
#QTrans.m4
cct_init
elen = 0.75
  Vs: source(up_ elen,S); llabel(-,V_s,+)
     resistor(right_ elen);  rlabel(,R_s)
     dot
     { capacitor(down_ to (Here,Vs.start))
       rlabel(+,V,-); llabel(,{1\over{j\omega C}},)
       dot }
    arrowline(right_ elen*2/3); llabel(,I)
  T1: transformer(down_ Vs.len,,6,,4) with .P1 at Here # Place P1
```

```
    "$T1$" at last [].n above
    line from T1.P2 to Vs.start
    line from T1.S1 up_ to (T1.S1,Vs.end) then right_ elen*2/3
    resistor(down_ Vs.len); rlabel(,R_L); b_current(I_L,rjust)
    line to (T1.S2,Here) then to T1.S2
.PE
```

Another composite element, `potentiometer(`*linespec*`,`*cycles*`,`*fractional pos*`,`*length*`, ...)`, shown in Figure 27, 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.

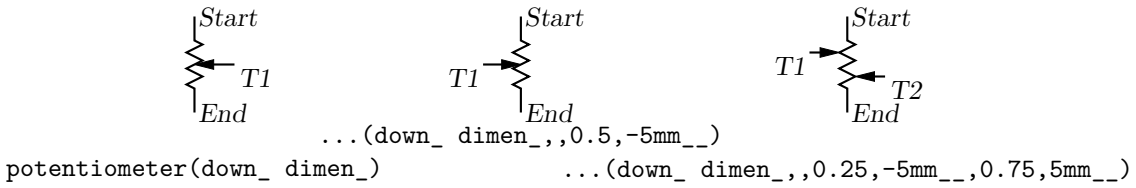


Figure 27: Default and multiple-tap potentiometer.

The macro `addtaps([`*arrowhd* `| type=`*arrowhd*`;name=`*Name*`],` *fraction, length, fraction, length,* ...), shown in Figure 28, will add taps to the immediately preceding two-terminal element.

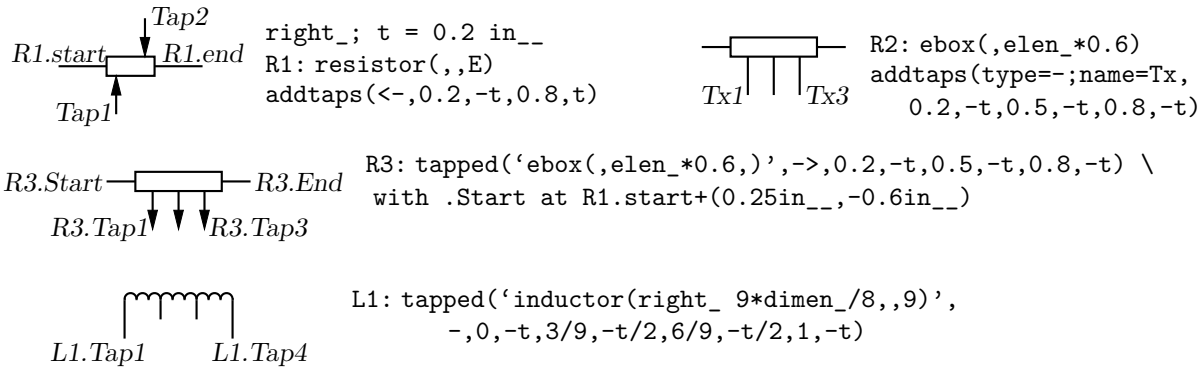


Figure 28: Macros for adding taps to two-terminal elements.

However, the default names `Tap1, Tap2` ... may not be unique in the current scope. An alternative name for the taps can be specified or, if preferable, the tapped element can be drawn in a [ ] block using the macro `tapped('`*two-terminal element*`', [`*arrowhd* `| type=`*arrowhd*`;name=`*Name*`],` *fraction, length, fraction, length,* ...). Internal names `.Start, .End,` and .C are defined automatically, corresponding to the drawn element. These and the tap names can be used to place the block. These two macros require the two-terminal element to be drawn either up, down, to the left, or to the right; they are not designed for obliquely drawn elements.

A few composite symbols derived from two-terminal elements are shown in Figure 29.

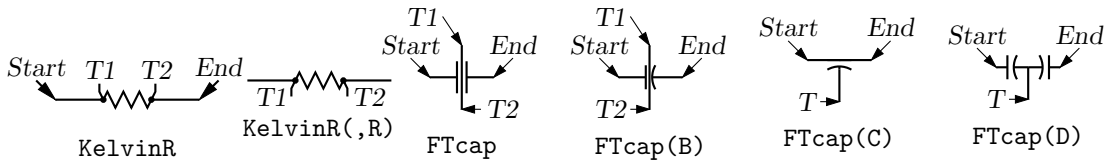


Figure 29: Composite elements `KelvinR(`*cycles*`,[R],`*cycle wid*`)` and `FTcap(`*chars*`)` .

The ground symbol is shown in Figure 30. The first argument specifies position; for example, `ground(at (1.5,2))` has the same effect as `move to (1.5,2); ground.` 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 D (down) the default. This macro is one of several in which a temporary drawing direction is set using the `setdir_(U|D|L|R|`*degrees*, *default* `R|L|U|D|`*degrees*`)` macro and reset at the end using `resetdir_`.

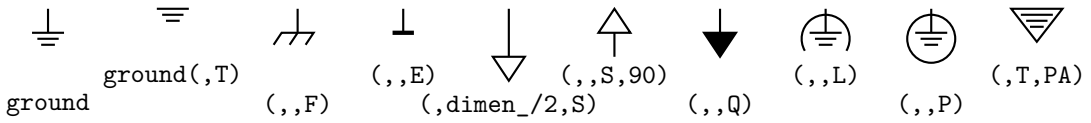


Figure 30: The `ground( at` *position*`, T|`*stem length*`, N|F|S|L|P[A]|E, U|D|L|R|`*degrees* `)` macro.

The arguments of `antenna(at` *position*`, T|`*stem length*`, A|L|T|S|D|P|F, U|D|L|R|`*degrees*`)` shown in Figure 31 are similar to those of `ground`.

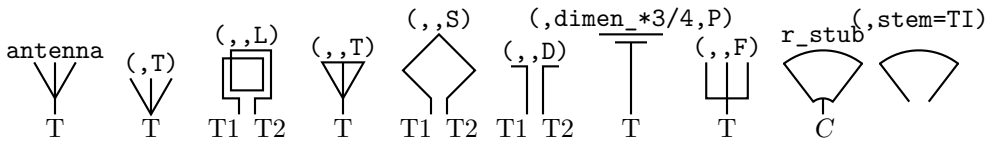


Figure 31: Antenna symbols with macro arguments shown above and terminal names below, and the microstrip radial stub `r_stub(at` *position, keys*`)`.

Figure 32 illustrates the macro `opamp(`*linespec, - label, + label, size, chars, attributes*`)`.

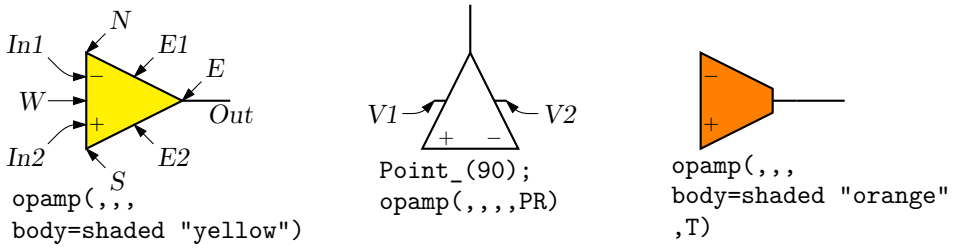


Figure 32: Operational amplifiers. The `P` option adds power connections. The second and third arguments can be used to place and rotate arbitrary text at `In1` and `In2`.

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 is a string of characters. `P` adds a power connection, `R` exchanges the second and third entries, and `T` truncates the opamp point.

Typeset text associated with circuit elements is not rotated by default, as illustrated by the second and third opamps in Figure 32. The `opamp` labels can be rotated if necessary by using postprocessor commands (for example PSTricks `\rput`) as second and third arguments.

The code in Figure 33 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
```

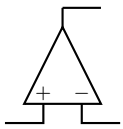


Figure 33: A code fragment invoking the `opamp(`*linespec*`,-,+,`*size*`,[R][P])` macro.

Figure 34 shows some audio devices, defined in `[]` blocks, with predefined internal locations as shown.

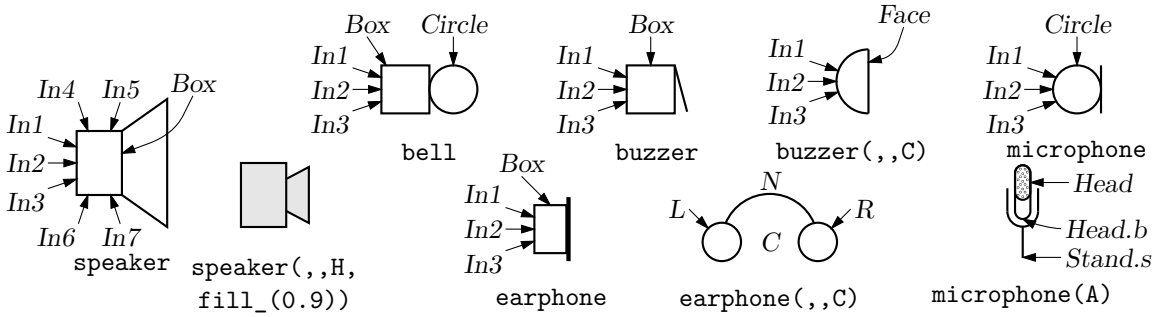


Figure 34: Audio components: `speaker(U|D|L|R|`*degrees*`,`*size*`,`*type*`,`*attributes*`), bell, microphone, buzzer, earphone`, with their internally named positions and components.

The first argument specifies the device orientation. The fourth can add fill or line attributes. Thus,

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

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

The `nport(`*box specs* `[;` *other commands*`]`, *nw, nn, ne, ns, space ratio, pin lgth, style*`)` macro is shown in Figure 35.

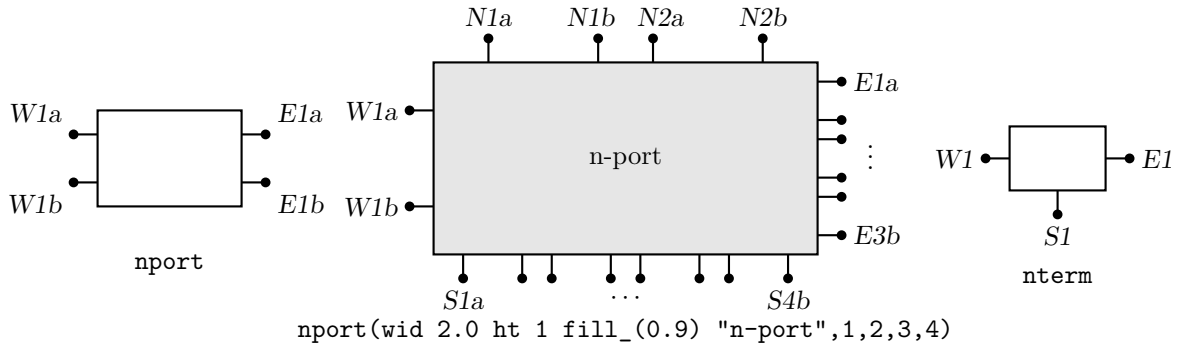


Figure 35: The `nport` macro draws a sequence of pairs of named pins on each side of a box. The pin names are shown. The default is a twoport. The `nterm` macro draws single pins instead of pin pairs.

The macro begins with the line `define(‘nport’,‘[Box: box ‘$1’`, so the first argument is a box specification such as size, fill, or text. The second to fifth arguments specify the number of ports (pin pairs) to be drawn respectively on the west, north, east, and south sides of the box. The end of each pin is named according to the side, port number, and *a* or *b* pin, as shown. The sixth argument specifies the ratio of port width to inter-port space, the seventh is the pin length, and setting the eighth argument to `N` omits the pin dots. The macro ends with `‘$9’]’)`, so that a ninth argument can be used to add further customizations within the enclosing block.

The `nterm(`*box specs, nw, nn, ne, ns, pin lgth, style*`)` macro illustrated in Figure 35 is similar to the `nport` macro but has one fewer argument, draws single pins instead of pin pairs, and defaults to a 3-terminal box.

Many custom labels or added elements may be required, particularly for 2-ports. These elements can be added using the first argument and the ninth of the `nport` macro. For example, the following code adds a pair of labels to the box immediately after drawing it but within the enclosing block:

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

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

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

The above example and the related gyrator macro are illustrated in Figure 36.

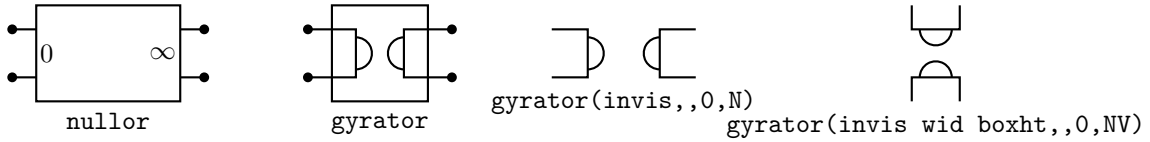


Figure 36: The `nullor` example and the `gyrator` macro are customizations of the `nport` macro.

Figure 37 shows the macro `contact(`*chars*`)`, which contains predefined locations *P*, *C*, *O* for the armature and normally closed and normally open terminals. An `I` in the first argument draws open circles for contacts.

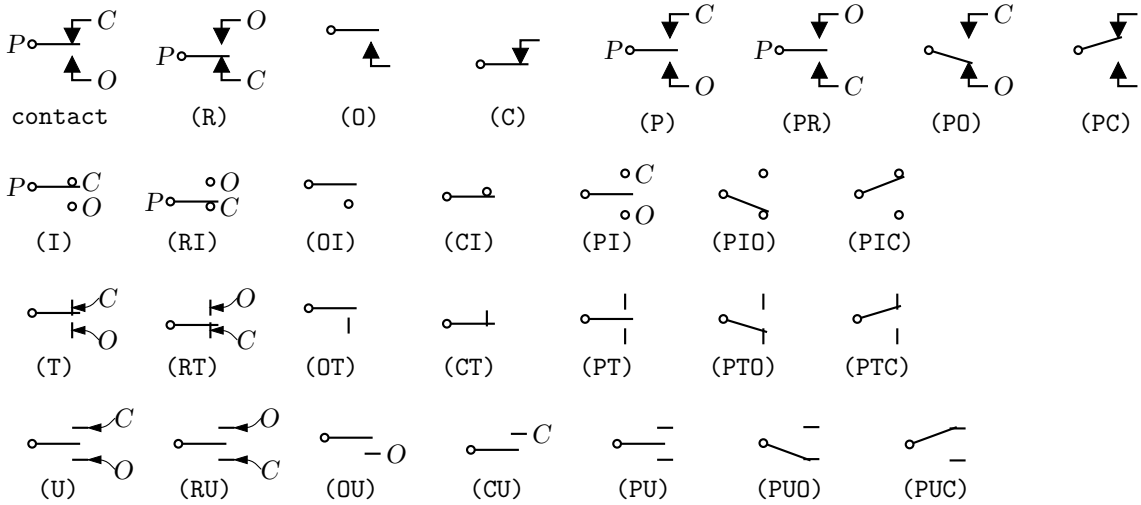


Figure 37: The `contact(`*chars*`)` macro (default drawing direction right) can be used alone, in a set of ganged contacts, or in relays.

The `contacts(`*poles*, *chars*`)` macro in Figure 38 draws multiple contacts.

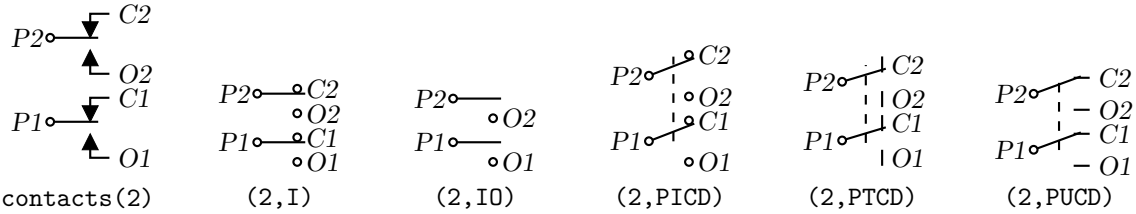


Figure 38: The `contacts(`*poles*, *chars*`)` macro (drawing direction right).

For drawing relays, the macro `relaycoil(`*chars*, *wid*, *ht*, `U|D|L|R|`*degrees*`)` shown in Figure 39 provides a choice of connection points and actuator types.

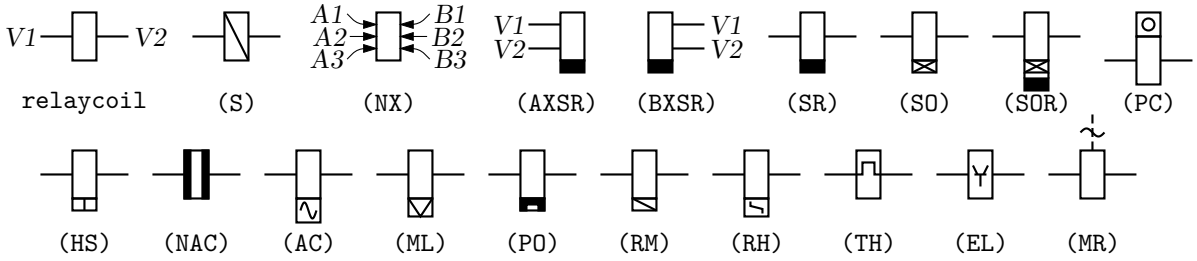


Figure 39: The `relaycoil` macro.

The `relay` macro in Figure 40 defines coil terminals *V1*, *V2* and contact terminals P_i, C_i, O_i.

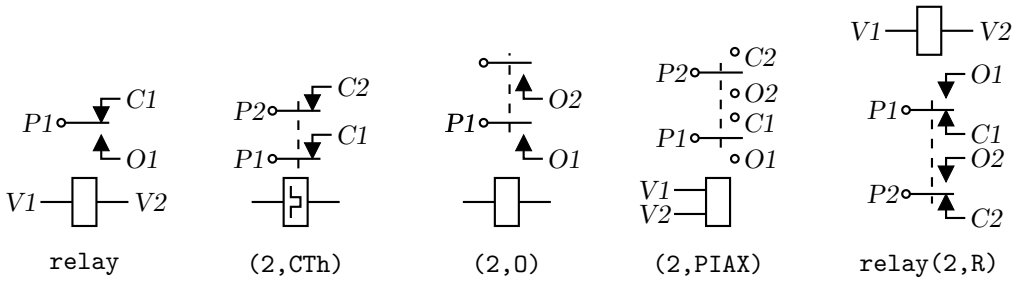


Figure 40: The `relay(`*poles, chars, attributes*`)` macro (drawing direction right).

The double-throw switches shown in Figure 41 are drawn in the current drawing direction like the two-terminal elements, but are composite elements that must be placed accordingly.

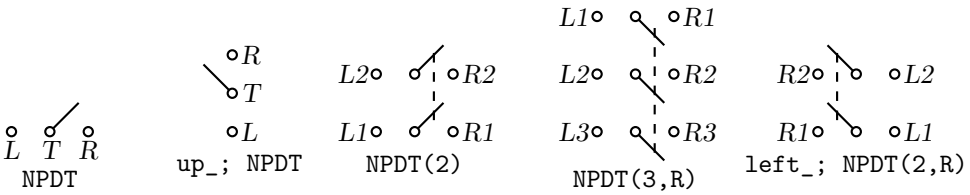


Figure 41: Multipole double-throw switches drawn by `NPDT(`*npoles*`, [R])`.

Figure 42 shows IEEE standard [8] examples plus a customization produced by the macro `RotarySwitch(` *start degrees*`:`*end degrees, keys* `)`. The poles are drawn in a counter-clockwise arc.

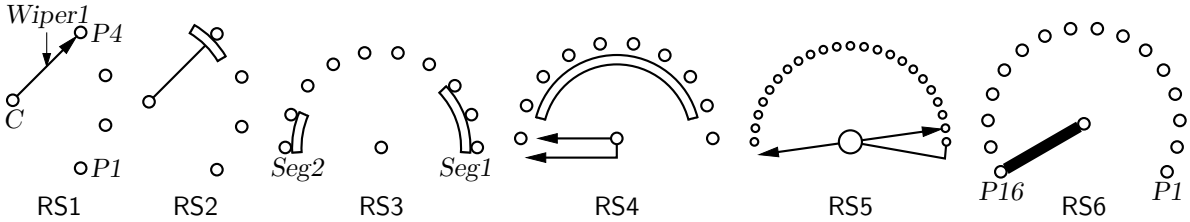


Figure 42: Rotary switches. Each instance contains position *C*, poles *P1*, *P2*,..., wipers *Wiper1*, ..., and segments *Seg1*, ... as required. Sources for the figure are in `RotarySwitch.m4` in the examples directory.

The `jack` and `plug` macros and their defined points are illustrated in Figure 43. The first argument of both macros establishes the drawing direction.

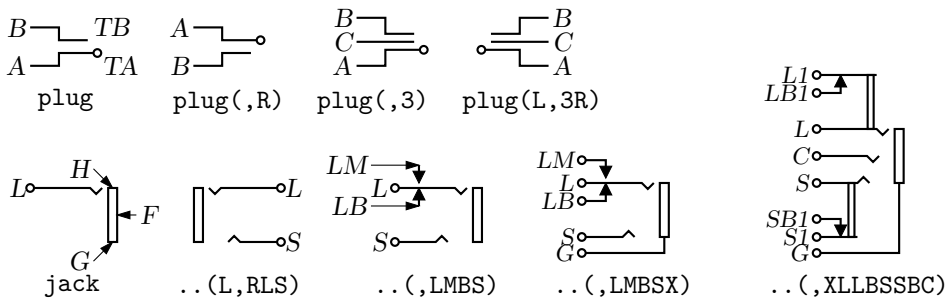


Figure 43: The `jack(U|D|L|R|`*degrees*`, `*chars*` [;`*keys*`])` and `plug(U|D|L|R|`*degrees*`,[2|3][R])` components and their defined points.

The second argument is a string of characters defining drawn components. An `R` in the string specifies a right orientation with respect to the drawing direction. The two principal terminals of the jack are included by putting `L S` or both into the string with associated make (`M`) or break (`B`) points. Thus, `LMB` within the third argument draws the `L` contact with associated make and break points. Repeated `L[M|B]` or `S[M|B]` substrings add auxiliary contacts with specified make or break points.

A macro for drawing headers is in Figure 44.

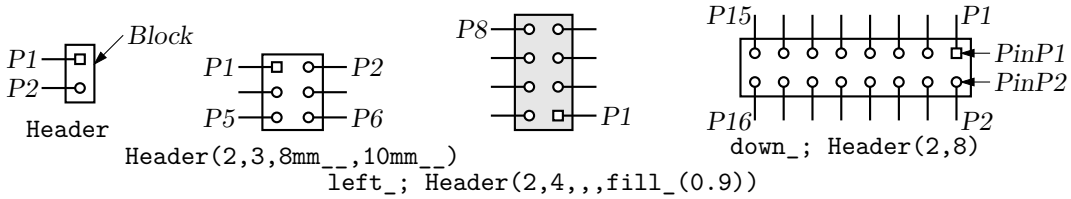


Figure 44: Macro `Header(1|2,` *rows, wid, ht, type*`)`.

Some connectors are shown in Figure 45, Figure 46, and Figure 47. The `tstrip` macro allows keys `wid=`*value*`;` `ht=`*value*`;` and `box=`*attributes*`;` in argument 3 for width, height, and e.g., fill, color, or dashed.

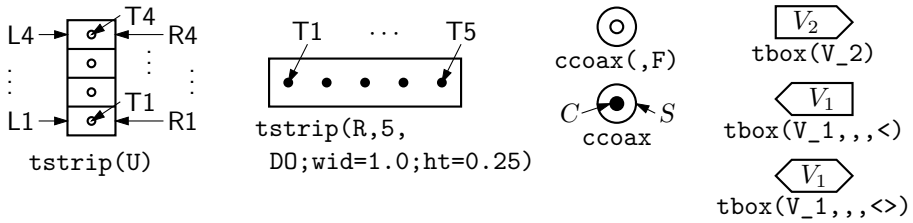


Figure 45: Macros `tstrip(R|L|U|D|`*degrees*`, `*chars*`)`, `ccoax(at` *location*`, M|F,` *diameter*`)`, and `tbox(`*text, wid, ht*`, <|>|<>,`*type, direction*`)`.

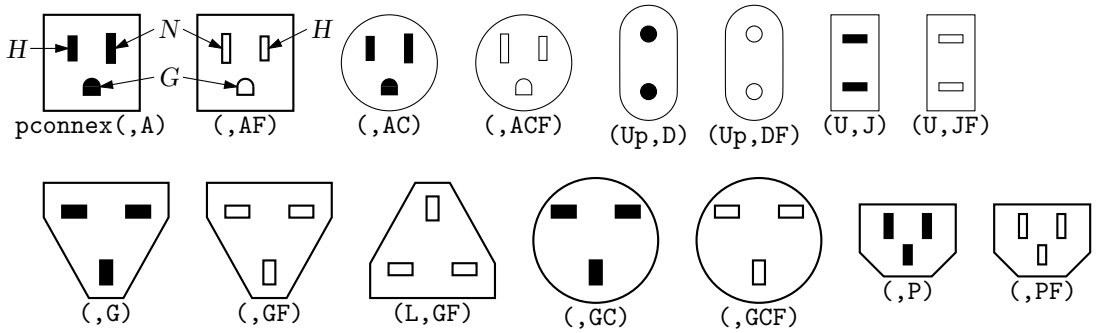


Figure 46: A small set of power connectors drawn by `pconnex(R|L|U|D|`*degrees, chars*`)`. Each connector has an internal `H`, `N`, and where applicable, a `G` shape.

The `tconn` macro draws a single pin or socket connector of a variety of shapes. These often occur in groups as drawn by the `Pconn` macro.

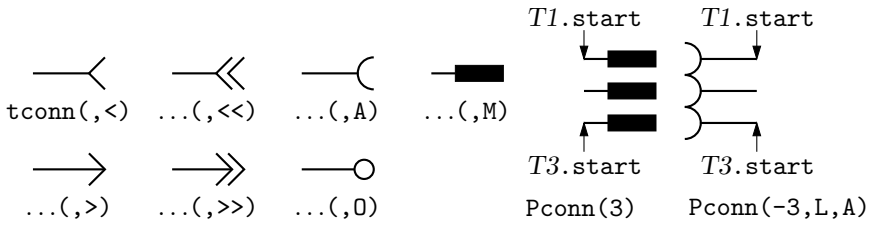


Figure 47: Macros `tconn(`*linespec*`,` *chars*`|`*keys*`,` *wid*`)` and `Pconn([−]`*n*`,U|D|L|R|`*degrees*`[:`*length*`],` *chars*`|`*keys*`)`. The latter draws `tconn` connectors in parallel groups.

A few compositie macros have no terminals at all. `ACsymbol` and `DCsymbol` have been mentioned; some others are `Ysymbol` and `Deltasymbol` with variations, `adjust`, and the `heatsink` shown in Figure 48.



Figure 48: The elements `ACsymbol`, `DCsymbol`, `Ysymbol`, `Ysymbol(,type=Z)`, `Deltasymbol`, `Deltasymbol(,type=E)`, `Deltasymbol(,type=O)`, `adjust`, and `heatsink(at` *position*, *keys*, `U|D|L|R|`*degrees*`)` have similar argument sequences.

A basic winding macro for magnetic-circuit sketches and similar figures is shown in Figure 49. For simplicity, the complete spline is first drawn and then blanked in appropriate places using the background (core) color (`lightgray` for example, default `white`).

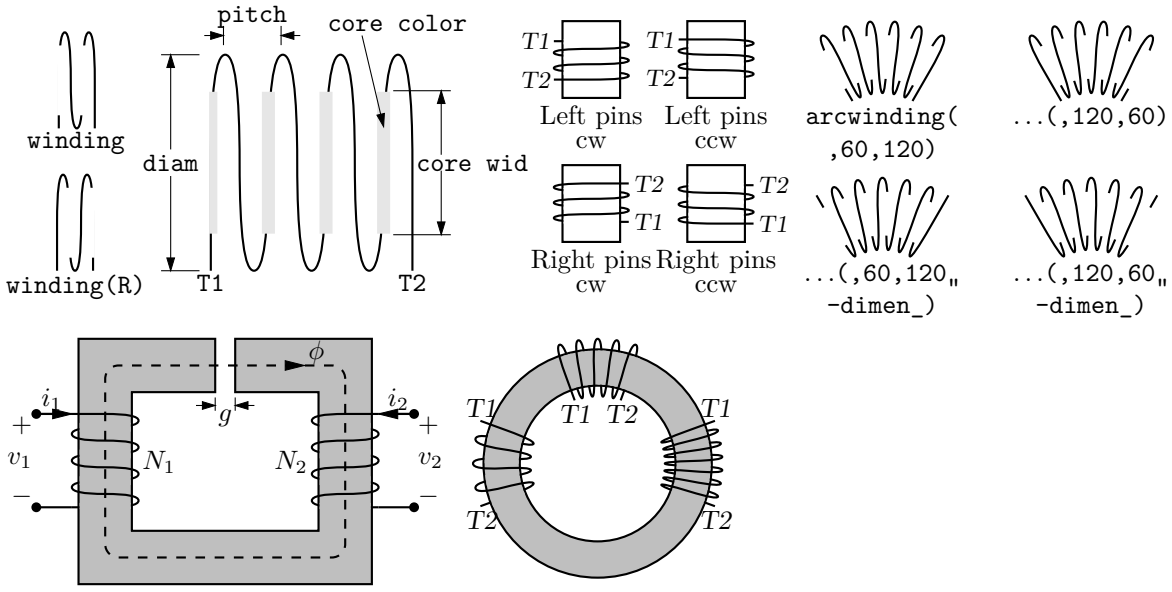


Figure 49: The `winding(`*L*/*R*, *winding diam*, *pitch*, *nturns*, *core wid*, `"`*core color*`")` macro draws a coil with axis along the current drawing direction. Terminals `T1` and `T2` are defined. Setting the first argument to `R` draws a right-hand winding. Similarly, `arcwinding(`*winding diam*, *start degrees*, *end degrees*, *nturns*, *core centre rad*, *core width*, `"`*core color*`")` draws curved windings. Negative *core centre rad* puts the terminals on the outside.

6.1 Semiconductors

Figure 50 shows the variants of bipolar transistor macro `bi_tr(`*linespec*`,L|R,P,E)` which contains predefined internal locations *E*, *B*, *C*.

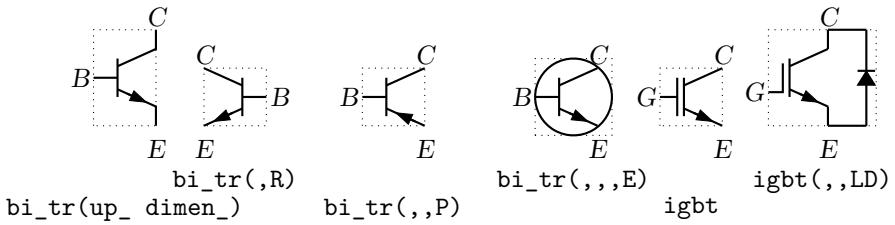


Figure 50: Variants of bipolar transistor `bi_tr(`*linespec*`,L|R,P,E)` (current direction upward).

The first argument defines the distance and direction from *E* to *C*, with location determined by the enclosing block as for other elements, and the base placed to the left or right of the current drawing direction according to the second argument. Setting the third argument to `P` creates a PNP device instead of NPN, and setting the fourth to `E` draws an envelope around the device.

Figure 51 shows a composite macro with several optional internal elements.

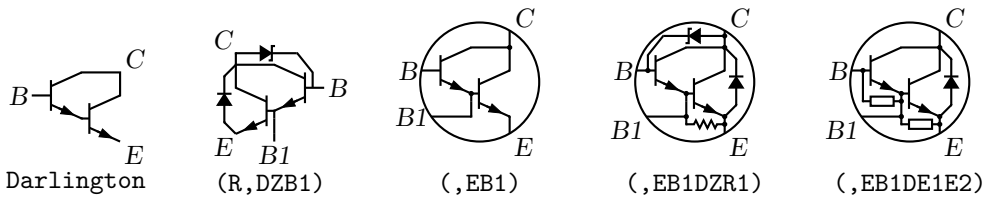


Figure 51: Macro `Darlington(L|R,[E][P][B1][E1|R1][E2|R2][D][Z])`, drawing direction `up_`.

The code fragment example in Figure 52 places a bipolar transistor, connects a ground to the emitter, and connects a resistor to the collector.

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

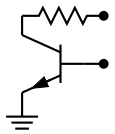


Figure 52: The `bi_tr(`*linespec*`,L|R,P,E)` macro.

The `bi_tr` and `igbt` macros are wrappers for the macro `bi_trans(`*linespec*`, L|R, `*chars*`, E)`, which draws the components of the transistor according to the characters in its third argument. For example, multiple emitters and collectors can be specified as shown in Figure 53.

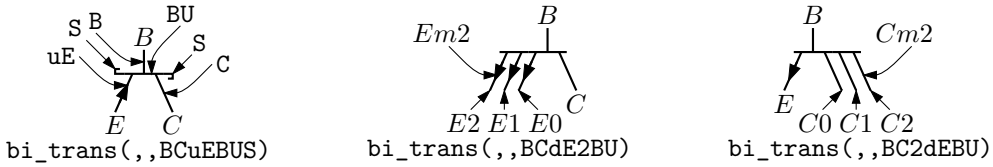


Figure 53: The `bi_trans(`*linespec*`,L|R,`*chars*`,E)` macro. The sub-elements are specified by the third argument. The substring `E`*n* creates multiple emitters *E0* to *En*. Collectors are similar.

A UJT macro with predefined internal locations *B1*, *B2*, and *E* is shown in Figure 54.

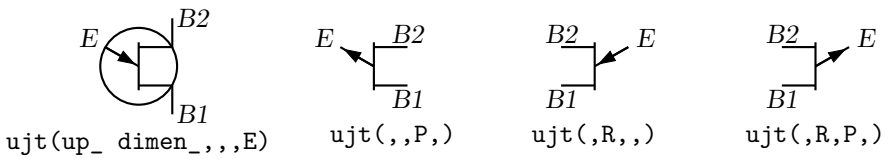


Figure 54: UJT devices, with current drawing direction `up_`.

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 55.

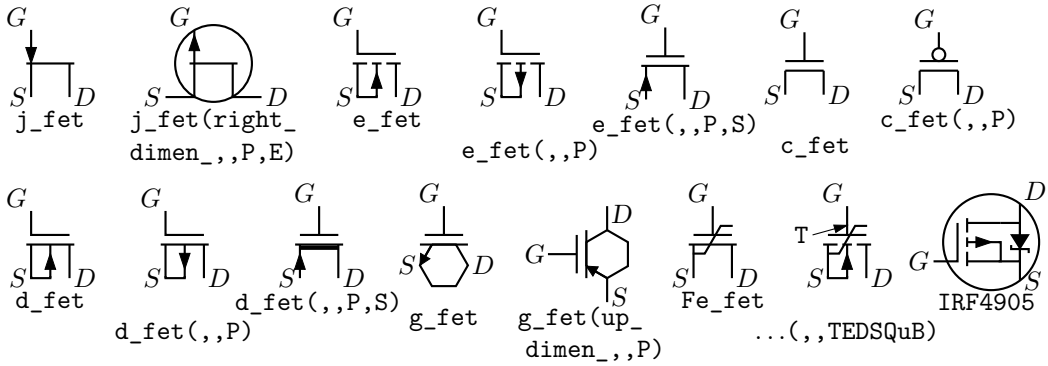


Figure 55: JFET, insulated-gate enhancement and depletion MOSFETs, simplified versions, graphene, ferroelectric fets, and a custom component. These macros are wrappers that invoke the `mosfet` macro.

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 figure are wrappers for the general macro `mosfet(`*linespec*`,R,`*characters*`,E)`. The third argument of this macro is a subset of the characters `BDEFGHKMPOQRSTXZ`, each letter corresponding to a diagram component as shown in the figure. Preceding the characters `B`, `G`, and `S` by `u` or `d` adds an up or down arrowhead to the pin, preceding `T` by `d` negates the pin, and preceding `M` by `u` or `d` puts the pin at the drain or source end respectively of the gate. This system allows considerable freedom in choosing or customizing components, and Figure 56 shows the subcomponents defined in `mosfet` together with some custom elements that could be put in wrappers if used often.

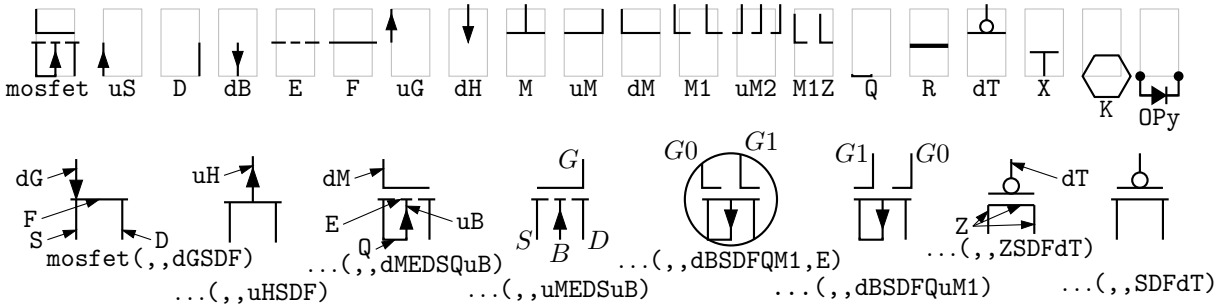


Figure 56: Subcomponents defined in the `mosfet` macro with a reference frame, showing some effects of preceding the subcomponent letter by `u` or `d`. The bottom-row contains custom devices.

The 3 or 4-terminal thyristor macro with predefined internal locations *G* and *T1*, *T2*, or *A*, *K*, *G*, and *Ga* as appropriate is in Figure 57. Except for the *G* and *Ga* terminals, most thyristor variants are much like a two-terminal element but a few are based on the `bi_trans` macro. The wrapper `thyristor_t(`*linespec, chars, label*`)` and similar macros `scr, scs, sus, sbs, gto` and `gts` place

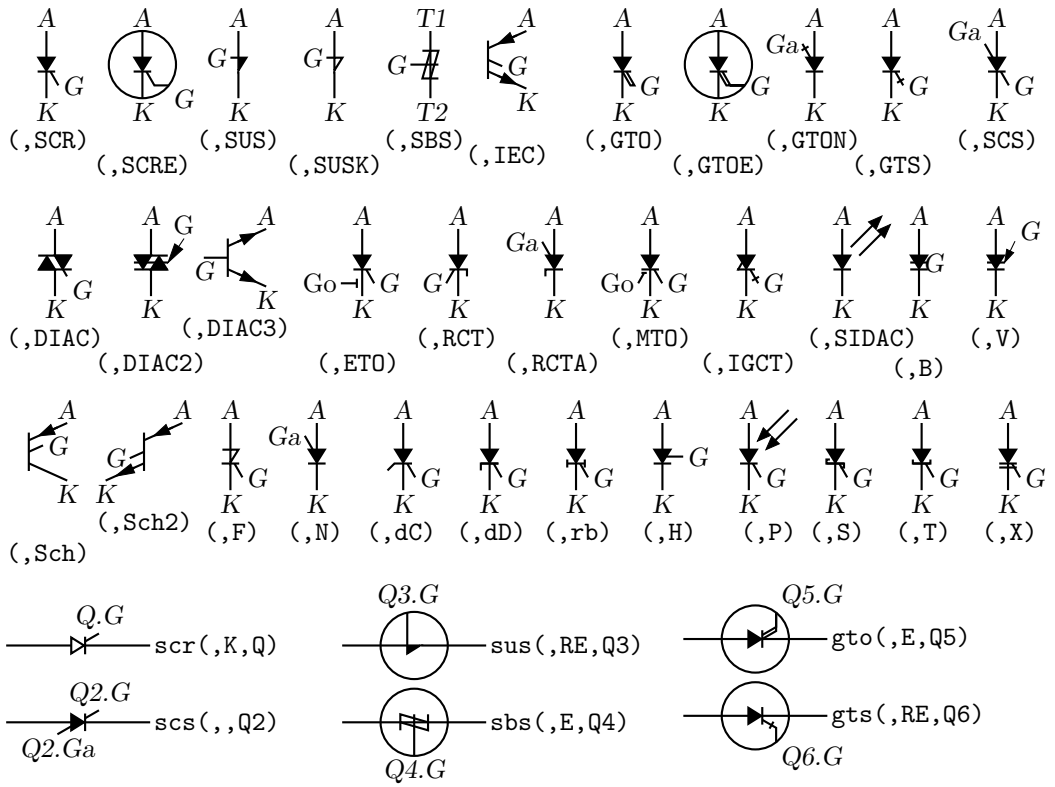


Figure 57: The top three rows illustrate use of the many-optioned `thyristor(`*linespec*, *chars*`)` macro, drawing direction `down_`, and the bottom rows show wrapper macros (drawing direction `right_`) that place the thyristor like a two-terminal element. Append `K` to the second argument to draw open arrowheads. The variations correspond to those defined for the `diode` macro.

thyristors using *linespec* as for a two-terminal element, but require a third argument for the label for the compound block; thus,

```
scr(from A to B,,Q3); line right from Q3.G
```

draws the element from position A to position B with label *Q3*, and draws a line from G.

The number of possible semiconductor symbols is very large, so these macros must be regarded as prototypes. Often an element is a minor modification of existing elements. The `thyristor(`*linespec*`,` *chars*`)` macro in Figure 57 is derived from diode and bipolar transistor macros. Another example is the `tgate` macro shown in Figure 58, which also shows a pass transistor.

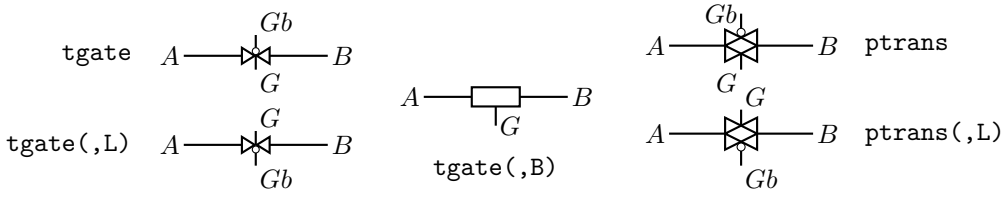


Figure 58: The `tgate(`*linespec*`, [B][R|L])` element, derived from a customized diode and `ebox`, and the `ptrans(`*linespec*`, [R|L])` macro. These are not two-terminal elements, so the *linespec* argument defines the direction and length of the line from A to B but not the element position.

Some other non-two-terminal macros are `dot`, which has an optional argument "`at` *location*", the line-thickness macros, the `fill_` macro, and `crossover`, which is a useful if archaic method to show non-touching conductor crossovers, as in Figure 59.

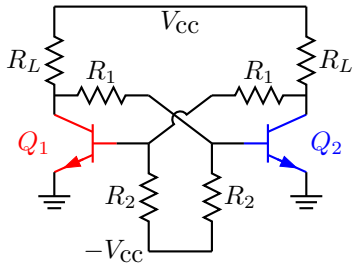


Figure 59: Bipolar transistor circuit, illustrating `crossover` and colored elements.

This figure also illustrates how elements and labels can be colored using the macro
`rgbdraw(`*r*`, `*g*`, `*b*`, `*drawing commands*`)`
where the *r*, *g*, *b* values are in the range 0 to 1 to specify the rgb color. This macro is a wrapper for the following, which may be more convenient if many elements are to be given the same color:
`setrgb(`*r*`, `*g*`, `*b*`)`
drawing commands
`resetrgb`
A macro is also provided for colored fills:
`rgbfill(`*r*`, `*g*`, `*b*`, `*drawing commands*`)`
These macros depend heavily on the postprocessor and are intended only for PSTricks, Ti*k*z PGF, MetaPost, SVG, and the Postscript or PDF output of dpic. Their effects are fragile in some situations. Basic Pic objects are probably best colored and filled as discussed in Section 3.4.

7 Corners

If two straight lines meet at an angle then, depending on the postprocessor, the corner may not be mitred or rounded unless the two lines belong to a multisegment line, as illustrated in Figure 60.

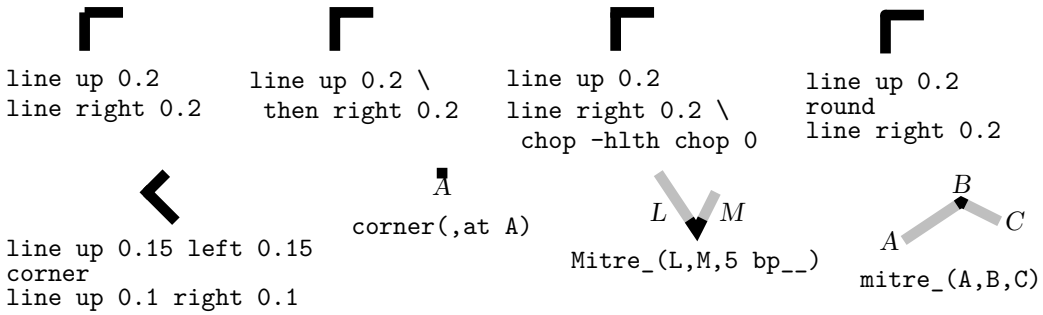


Figure 60: Producing mitred angles and corners.

This detail is normally not an issue for circuit diagrams unless the figure is magnified or thick lines are drawn. Rounded corners can be obtained by setting post-processor parameters, but the figure shows the effect of macros `round` and `corner`. The macros `mitre_(`*Position1, Position2, Position3, length, attributes*`)` and `Mitre_(`*Line1, Line2, length, attributes*`)` may assist as shown. Otherwise, a right-angle line can be extended by half the line thickness (macro `hlth`) as shown on the upper row of the figure, or a two-segment line can be overlaid at the corner to produce the same effect.

8 Looping

Sequential actions can be performed using either the dpic command

`for` *variable*`=`*expression* `to` *expression* `[by` *expression*`] do {` *actions* `}`

or at the m4 processing stage, which is executed and finished before dpic or gpic begins. An m4 macro inside a pic loop is expanded only once and the resulting expansion is executed with each pic repetition. As an alternative, the `libgen` library defines the m4 macro

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

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

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

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

(an alias for the older macro `Loopover_`), which evaluates *actions* and increments counter `m4Lx` for each instance of *variable* set to *value1, value2, ...*

9 Logic gates

Library `liblog.m4` contains a selection of basic and advanced logic gates and structures. The default size and style parameters defined near the top of the file can be globally redefined or temporarily set locally. Individual gates also have arguments that allow adjustment of size, and fill, for example.

Figure 61 shows the basic logic gates. The first argument of the gate macros can be an integer N from 0 to 16, specifying the number of input locations `In1, ... 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. Alternately, the first argument can be a sequence of letters `P` or `N` to define a number of normal or negated (`N`ot-circled) inputs.

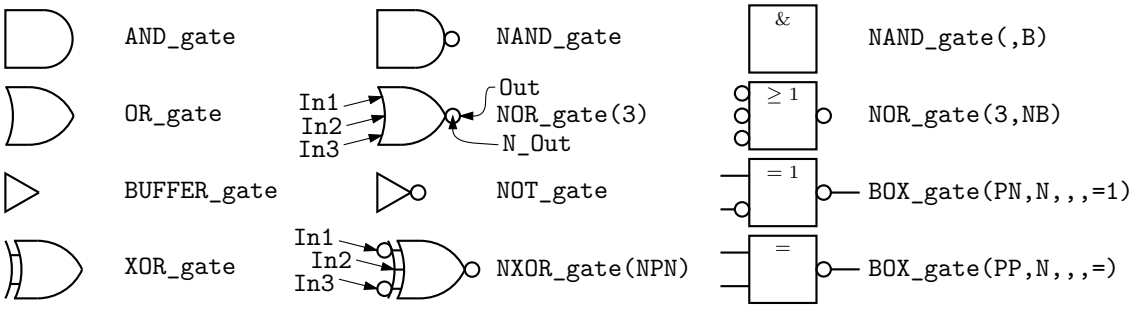


Figure 61: Basic logic gates. The input and output locations of a three-input NOR gate are shown. Inputs are negated by including an `N` in the second argument letter sequence. A `B` in the second argument produces a box shape as shown in the rightmost column, where the second example has AND functionality and the bottom two are examples of exclusive OR functions.

Inputs retain their positions relative to the body regardless of gate orientation, as in Figure 62.

```
.PS
# ‘FF.m4’
log_init
Sg: NOR_gate
  left_
Rg: NOR_gate at Sg+(0,-L_unit*(AND_ht+1))
  line from Sg.Out right L_unit*3 then down Sg.Out.y-Rg.In2.y then to Rg.In2
  line from Rg.Out left L_unit*3 then up Sg.In2.y-Rg.Out.y then to Sg.In2
  line left 4*L_unit from Sg.In1 ; "$S$" rjust
  line right 4*L_unit from Rg.In1 ; "$R$" ljust
.PE
```

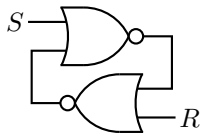


Figure 62: *SR* flip-flop.

Beyond a default number (6) of inputs, the gates are given wings as in Figure 63.

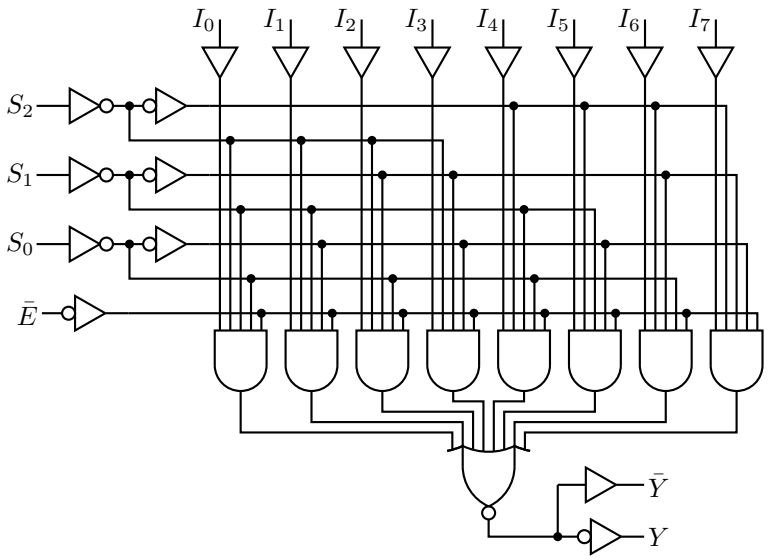


Figure 63: Eight-input multiplexer, showing a gate with wings.

Negated inputs or outputs are marked by circles drawn using 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`. Instead of a number, the first argument can be a sequence of letters `P` or `N` to define normal or negated inputs; thus for example, `NXOR_gate(NPN)` defines a 3-input nxor gate with not-circle inputs `In1` and `In3` and normal input `In2` as shown in Figure 61. The macro `IOdefs` can also be used to create a sequence of custom named inputs or 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_unit`s 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.

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 parameterized 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

The macro `BUFFER_gate(`*linespec*`,[N|B],`*wid*`,`*ht*`,[N|P]*,[N|P]*)` is a wrapper for the composite element `BUFFER_gen`. If the second argument is `B`, then a box gate is drawn; otherwise the gate is triangular. Arguments 5 and 6 determine the number of defined points along the northeast and southeast edges respectively, with an `N` adding a NOT circle. If the first argument is non-blank however, then the buffer is drawn along an invisible line like a two-terminal element, which is convenient sometimes but requires internal locations of the block to be referenced using `last []`, as shown in Figure 64.

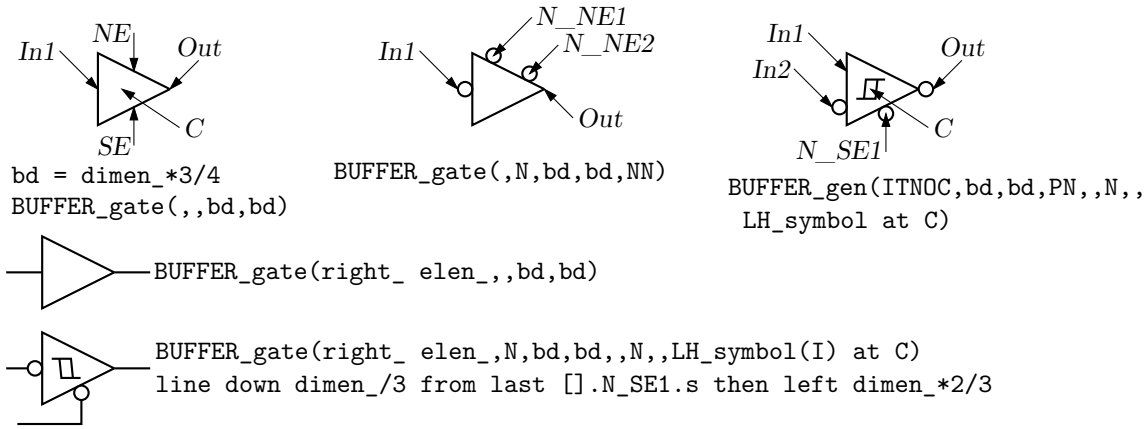


Figure 64: The `BUFFER_gate` and `BUFFER_gen` macros. The bottom two examples show how the gate can be drawn as a two-terminal macro but internal block locations must be referenced using `last []`.

Figure 65 shows the macro `FlipFlop(D|T|RS|JK,` *label, boxspec, pinlength*`)`, which is a wrapper for the more general macro `FlipFlopX(`*boxspec, label, leftpins, toppins, rightpins, bottompins, pinlength*`)`.

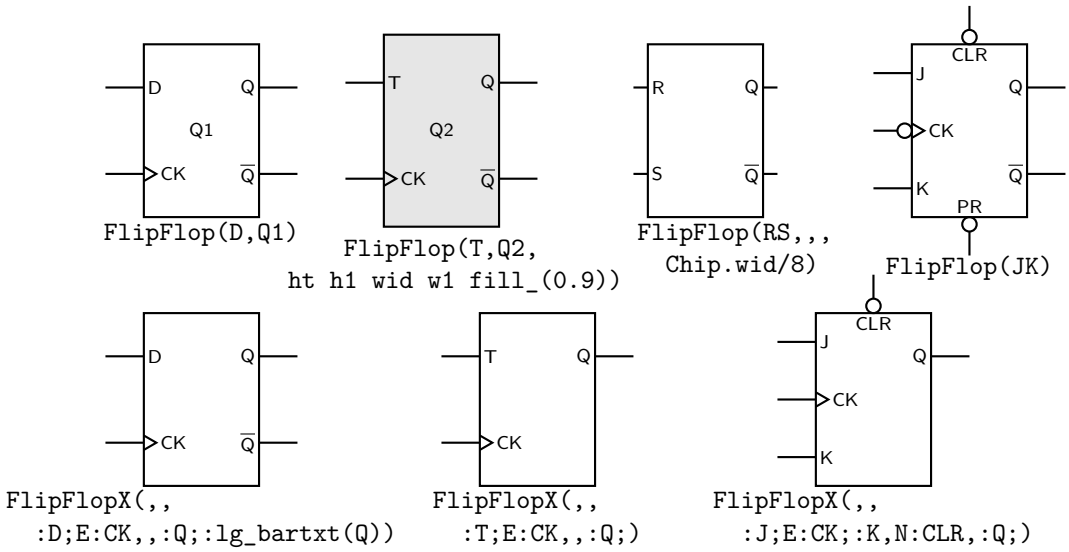


Figure 65: The `FlipFlop` and `FlipFlopX` macros, with variations.

The first argument modifies the box (labelled *Chip*) default specification. Each of arguments 3 to 6 is null or a string of *pinspecs* separated by semicolons (`;`). A *pinspec* is either empty (null) or of the form `[`*pinopts*`]:[`*label*`[:`*Picname*`]]`. The first colon draws the pin. Pins are placed top to bottom or left to right along the box edges with null pinspecs counted for placement. Pins are named by side and number by default; eg `W1, W2, ..., N1, N2, ..., E1, ..., S1, ...` ; however, if `:`*Picname* is present in a *pinspec* then *Picname* replaces the default name. A *pinspec* label is text placed at the pin base. Semicolons are not allowed in labels; use e.g., `\char59{}` instead. To put a bar over a label, use `lg_bartxt(`*label*`)`. The *pinopts* are `[L|M|I|O][N][E]` as for the `lg_pin` macro.

Optional argument 7 is the pin length in drawing units.

Figure 66 shows a multiplexer block with variations, and Figure 67 shows the very similar demultiplexer.

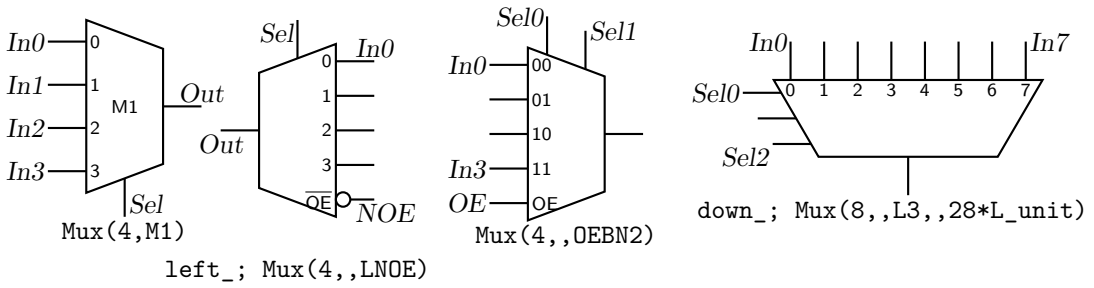


Figure 66: The `Mux(`*input count*`,` *label*`, [L][B|H|X][N[`*n*`]|S[`*n*`]][[N]OE],`*wid*`,`*ht*`)` macro.

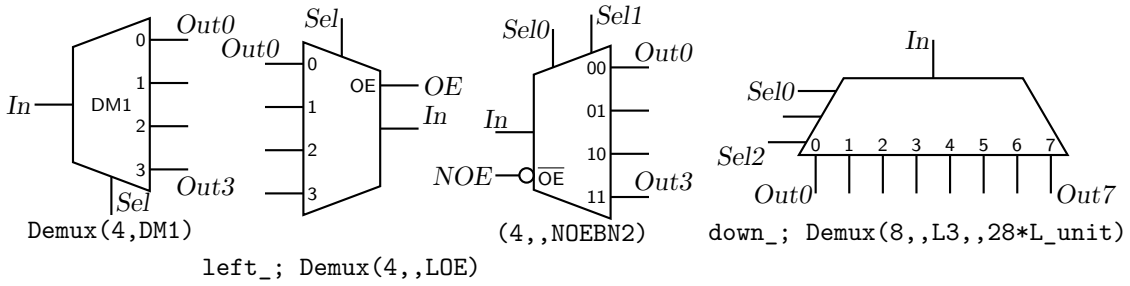


Figure 67: The `Demux(`*input count*`,` *label*`, [L][B|H|X][N[`*n*`]|S[`*n*`]][[N]OE],`*wid*`,`*ht*`)` macro.

Customized gates can be defined simply. For example, the following code defines the custom flipflops in Figure 68.

```
define(‘customFF’,‘FlipFlopX(wid 10*L_unit ht FF_ht*L_unit,,
    :S;NE:CK;:R, N:PR, :Q;;ifelse(‘$1’,1,:lg_bartxt(Q)), N:CLR) ’)
```

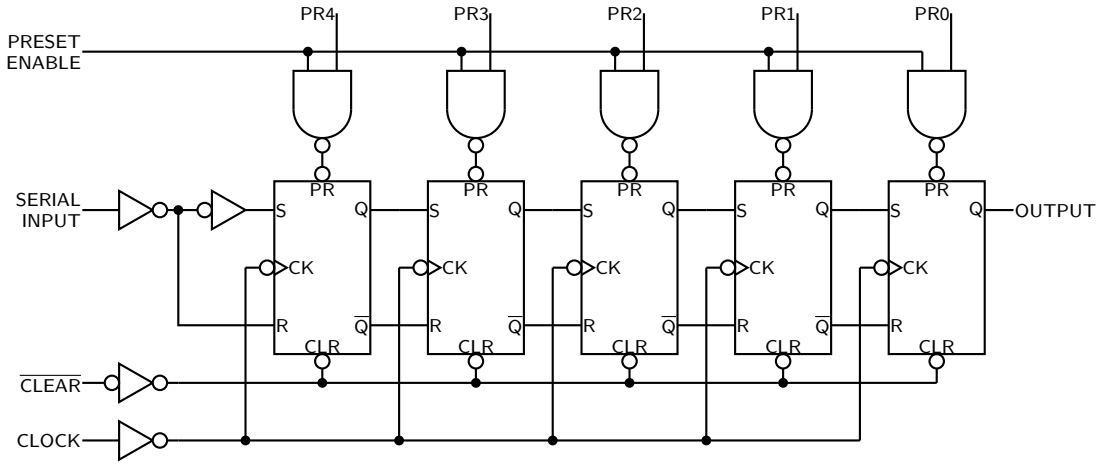


Figure 68: A 5-bit shift register.

This definition makes use of macros `L_unit` and `FF_ht` that predefine default dimensions. There are three pins on the right; the centre pin is null and the bottom is null if the first macro argument is 1.

For hybrid applications, the `dac` and `adc` macros are illustrated in Figure 69. The figure shows the default and predefined internal locations, the number of which can be specified as macro arguments.

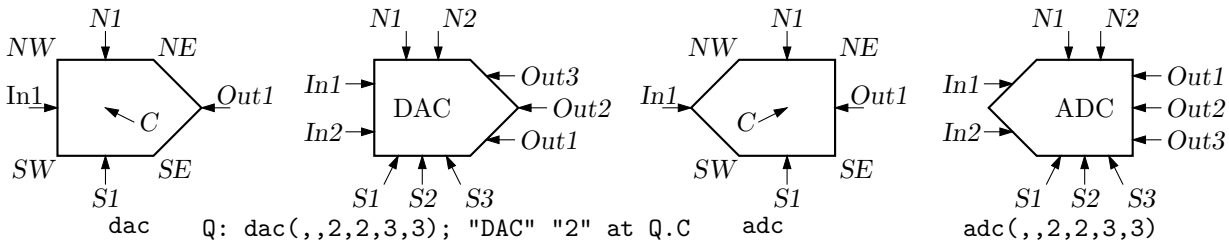


Figure 69: The `dac(`*width,height*`,nIn,nN,nOut,nS)` and `adc(`*width,height*`,nIn,nN,nOut,nS)` macros.

In addition to the logic gates described here, some experimental IC chip diagrams are included with the distributed example files.

9.1 Automatic structures

In some common but special cases, logic circuits having a predefined structure can be drawn automatically, thereby saving much repetitive code. Boolean functions expressed as a product of sums or a sum of products are examples, and result in two-layer diagrams. Consider for example, the function

$$f(a,b,c,d) = abcd + \sim ba + c + b\sim a$$

which is the sum (that is, "or") of four terms which are products (that is, "and") of one or more single-character variables or their negation indicated by a preceding tilde. This and similar functions can be drawn in two-layer form, as follows. Define the circuit using function notation with the logic-gate functions `And, Or, Not, Buffer, Xor, Nand, Nor,` and `Nxor.` Variables can also be negated using tilde notation as shown above. An m4 macro implementing a stack can parse the defining function and draw the corresponding structure, as shown in Figure 70 for the above example.

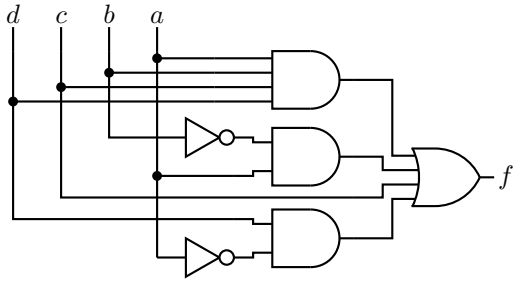


Figure 70: The circuit drawn by `Autologix(Or(And(a,b,c,d),And(Not(b),a),c,And(d,Not(a))))`.

Such an implementation is the macro

`Autologix(`*function-spec*`;` *function-spec*`; ..., [M[irror]] [N[occonect]] [L[eftinputs]] [R][V] [;offset=`*value*`])`

where *function-spec* is of the form *function*`(`*args*`)` `[@` *location-attribute*`]`, e.g.,

`HalfAdder: Autologix(Xor(x,y);And(x,y),LVR).`

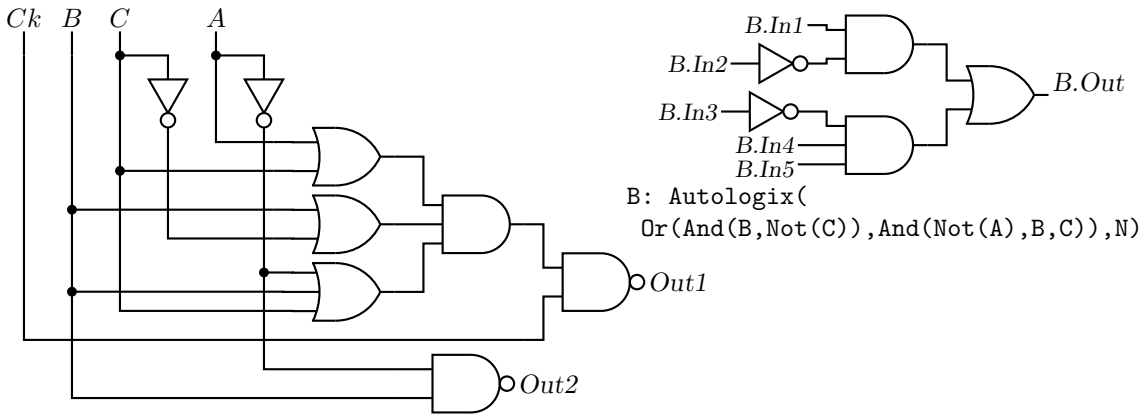
This macro draws one or more trees of gates with the output or outputs (treeroots) to the right (on the left if the `M[irror]` option is used). The predefined functions are given above and may be nested; e.g., `Or(And(x,~y),And(~x,y)).` The output is contained in a `[ ]` block, which can be positioned normally. Function notation does not model internal connections such as feedback, however, but internal nodes can be accessed and connections added.

The resulting block has outputs labeled *Out1, Out2, ...* corresponding to the functions in the first argument, and inputs labeled *In*<*var*> for each variable <*var*> in the defining expressions, (with NOT gates for variables preceded by `~`).

The exact appearance of a tree depends on the order in which terms and variables appear in the expressions. Gates can be placed relative to previously drawn objects using the `@` *location* construct; e.g., `@with .nw at last [].sw+(0,-dimen_).`

The macro has option `R` for reversing the drawn order of the inputs `N` for omitting input connections, and `V` to reverse the order in which variables are scanned. There is also a limited capability `L` for drawing inputs on the left; their vertical placement can be adjusted by adding `;offset=`*var*.

To assist in manually adding connections to the resulting structure, the internal gate inputs and outputs are defined and numbered *In1, In2, ...* and *Out1, Out2, ...*. These labels are listed at the end of the output of `Autologix`. Inputs are shown for an example in in Figure 71.




```
B: Autologix(
 Or(And(B,Not(C)),And(Not(A),B,C)),N)
```

```
Autologix(
  Nand(And(Or(A,C),Or(B,~C),Or(~A,B,C)),Ck);
  Nand(~A,B))@with .n at last [].s+(0,-2bp__))
```

Figure 71: The `Autologix(`*expression; expression;..., options*`)` macro automatically draws Boolean expressions in function notation. The function tree is drawn, then a row or column of inputs, then the connections. A default result is on the left, and a tree of gates without input connections but with internal input labels shown is at the upper right.

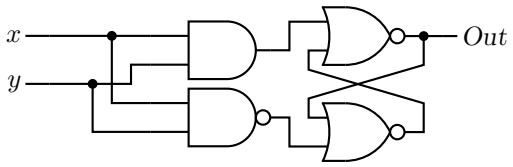
The given expressions need not be in canonical two-layer form and, with minor effort, custom gates beyond those mentioned above can be defined and included. Here is how to include an arbitrary circuit (an SR-flipflop, for example) that is not one of the standard gates. First, define the circuit with a name ending in `_gate`. Put its inputs named *In1, In2, ...* on the left and the output *Out* on the right:

```
define(‘SR_gate’,‘[ u = 2*L_unit
  S: NOR_gate
    line right_ 2*u from S.Out
  Out: Here
  R: NOR_gate at S+(0,-5*u)
  TS: S.In2-(u,0)
  TR: (TS,R.In1)
    dot(at S.Out+(u,0))
    line down u*3/2 then to TR+(0,u) then to TR then to R.In1
    line from R.Out right u then up u*3/2 then to TS+(0,-u) \
      then to TS then to S.In2
  In1: S.In1
  In2: R.In2 ]’)
```

Now define the function by which the circuit will be invoked using the built-in `_AutoGate` and the circuit name omitting `_gate`:

```
  define(‘SRff’,‘_AutoGate(SR,$@)’)
```

That is all. The result, with a NAND and an AND gate, is shown in Figure 72:




```
Autologix(SRff(And(x,y),Nand(x,y)),LRV)
```

Figure 72: The `SRff` example.

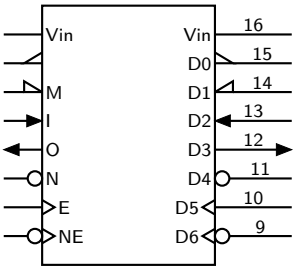
10 Integrated circuits

Developing a definitive library of integrated circuits is problematic because context may determine how they should be drawn. Logical clarity may require drawing a functional diagram in which the connection pins are not in the physical order of a terminal diagram, for example. Circuit boards and connectors are similar. Although the geometries are simple, managing lists of pin locations and labels can be tedious and repetitive.

The many-argument macro `lg_pin(` *location, label, Picname,* `n|e|s|w [L|M|I|O][N][E],` *pinno, optional length*`)` can be used to draw a variety of pins as illustrated in Figure 73. To draw the left-side pins, for example, one can write

```
lg_pin( U.nw-(0,lg_pinsep), Vin, Pin1, w )
lg_pin( U.nw-(0,2*lg_pinsep),,, wL )
```

and so on. Each pin can also be given a pic name, some text to indicate function, and a number but, to reduce the tedium of adding the pins by hand, a list can be given to `foreach_(‘variable’,` ‘*actions*’, *value1, value2, . . .*) which executes the given actions successively with *variable* = *value1*, *value2* . . . and the counter `m4Lx` set to 1, 2, ... The remaining left-side and the right-side pins in the figure have been specified using this macro.




```
.PS
# SampleIC.m4
log_init
command "\small\sf"

 U: box wid 18*L_unit ht 9*lg_pinsep

 lg_pin(U.nw-(0,lg_pinsep),Vin,Pin1,w)
 lg_pin(U.nw-(0,2*lg_pinsep),,,wL)

 foreach_(‘x’,
  ‘lg_pin(U.nw-(0,(m4Lx+2)*lg_pinsep),x,,w‘’x)’,
   M,I,O,N,E,NE)
 define(‘Upin’,
  ‘lg_pin(U.ne-(0,(17-‘$1’)*lg_pinsep),‘$2’,Pin‘$1’,e‘$3’,‘$1’,8*L_unit)’)
 foreach_(‘x’,
 ‘Upin(patsubst(x,;,‘,’))’,
  16;Vin;, 15;D0;L, 14;D1;M, 13;D2;I, 12;D3;O, 11;D4;N, 10;D5;E, 9;D6;NE )
.PE
```

Figure 73: An imaginary 16-pin integrated circuit and its code. Pin variations defined individually and by the first `foreach_` are shown on the left; and text, pic labels, and pin numbers are defined on the right. The third and successive arguments of the second `foreach_` are `;`-separated pin number, text, and pin type. The semicolons are changed to commas by the `patsubst` m4 macro and the `Upin` macro gives the resulting arguments to `lg_pin`.

11 Single-line diagrams

Standard single-line diagrams for power distribution employ many of the normal two-terminal elements along with others that are unique to the context. This distribution contains a library of single-line diagram (SLD) elements that can be loaded with the command `include(libSLD.m4)`. The `examples.pdf` and `examplesSVG.html` documents include samplers of some of their uses.

The SLD macros allow considerable scope for customization using key-value pairs to set internal parameters. In addition, diagram-wide or block-scope changes are made as usual by redefining environmental variables, particularly `linethick,` for example, and `linewidth` for scaling. Element body sizes are altered using, for example, `define(‘dimen_’,dimen_*1.2)` as for the normal circuit elements. To apply such a change to a single element or a group of them, use `pushdef(‘dimen_’,`*expr*`)` *element statements* `popdef(‘dimen_’).` The SLD library also includes a number of redefinable default style parameters, which are currently as follows:

```
define(‘sl_breakersize_’,‘dimen_*3/16’)  # breaker box size
define(‘sl_breakersep_’,‘dimen_/2’)      # breaker separation from body
define(‘sl_ttboxlen_’,‘dimen_*3/4’)      # inline box length
define(‘sl_ttboxwid_’,‘dimen_*3/4’)      # inline box width
define(‘sl_sboxlen_’,‘dimen_*2/3’)       # stem box length
define(‘sl_sboxwid_’,‘dimen_*2/3’)       # stem box wid
define(‘sl_diskdia_’,‘dimen_*2/3’)       # sl_disk diam
define(‘sl_chevronsiz_’,‘dimen_/4’)      # sl_drawout (chevron) size
define(‘sl_loadwid_’,‘dimen_*0.32’)      # load width
define(‘sl_loadlen_’,‘dimen_*0.45’)      # load length
define(‘sl_transcale_’,1)                # transformer body scale factor
define(‘sl_busthick_’,linethick*2)       # sl_bus line thickness
define(‘sl_busindent_’,‘min(dimen_/5,rp_len/5)’) # busbar end indent
```

The greatest control of appearance is obtained by drawing all elements individually; however, provision is made for automatically attaching circuit breakers (which occur often) and other symbols to elements.

11.1 Two-terminal SLD elements

The two-terminal SLD elements are drawn along an invisible line segment that can be named as for normal two-terminal elements. There are four arguments for which defaults are provided as always. The transformers are shown in Figure 74 and other two-terminal elements in Figure 75.

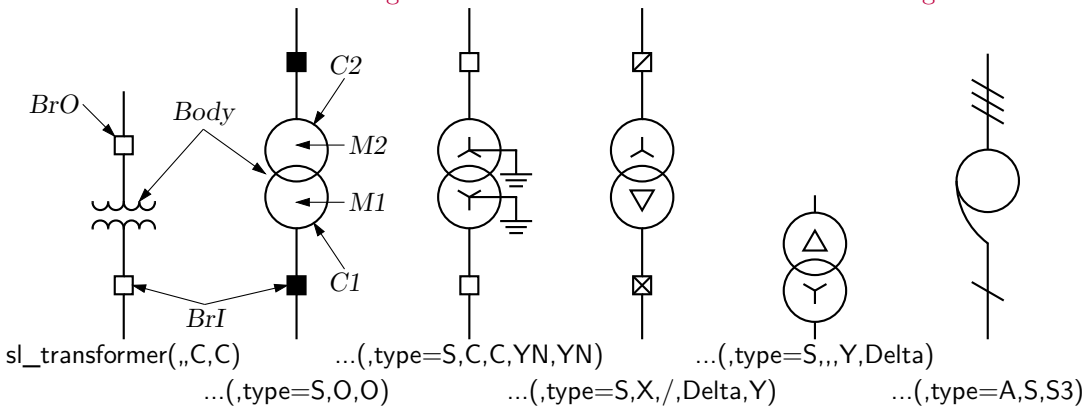


Figure 74: The SLD transformers drawn by `sl_transformer(`*linespec, key-value pairs, stem object, stem object, type* `S` *circle object, type* `S` *circle object*`)`, drawing direction `up_`.

The first argument is the linespec defining the direction and location of the element, e.g., `sl_transformer(right_` *expr*`).`

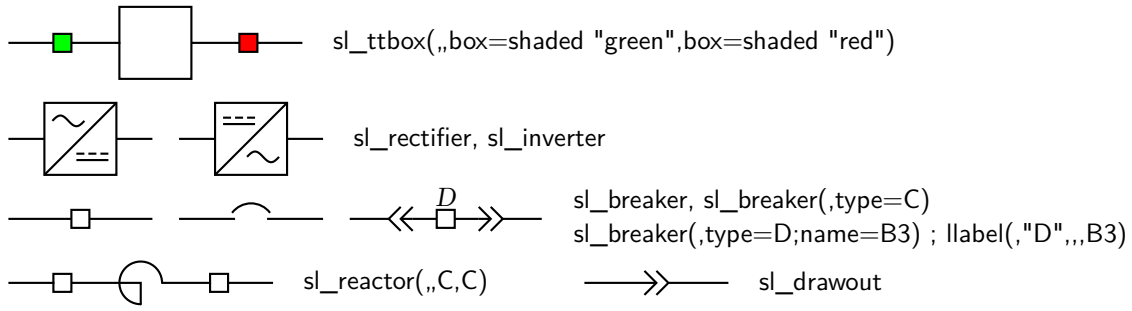


Figure 75: SLD two-terminal elements, drawing direction `right_`.

The second argument is a sequence of semicolon (;)-separated key-value pairs that customize the element body, depending on the case, e.g., `sl_ttbox(,lgth=`*expr*`; wdth=`*expr*`; text="`*internal label*`"; box=shaded "yellow").`

If the third argument is blank, then a plain input stem is drawn for the element. If it is a `C` then a default closed breaker is inserted and an `O` inserts a default open breaker, and similarly an `X` or slash (`/`) add these elements. If it or its prefix is `S:` or `S`*n*`:` where *n* is an integer, then, instead of a breaker, an *n*-line slash symbol is drawn using the macro `sl_slash(at` *position, keys,* `[`*n*`:]R|L|U|D|`*degrees*`).`

The separation of the optional attached breaker or other stem elements from the body is controlled by the `sl_breakersep_` global parameter. Adding `sep=`*expr* to the body keys adjusts separations for an element; otherwise, adding this key to argument 3 or 4 adjusts the separation of the corresponding attached object.

Otherwise, one or more of the extensive `sl_ttbox` body key-value pairs will insert a custom breaker as needed. These keys include: `lgth=`*expr*`, wdth=`*expr*`, name=`*Name*`, text="`*text*`", box=`*other box attributes, e.g.,* `dashed, shaded,` .... For the slash symbol, the `sl_slash` keys are valid.

The fourth argument is like the third but controls a breaker or slash symbol in the output lead. The example, `sl_transformer(right_ elen_ from A,,C,C)` draws a transformer with closed breakers in the input and output leads.

Exceptions are the `sl_drawout()` element which does not have breakers and the `transformer()` element which has an extra two arguments for the frequently used `S` variant.

The body can be given a name with `name=`*Label*`;` in the second argument. The default two-terminal name is *Body* except for the `sl_breaker` element which has default body name *Br* and the `sl_slash` element which has default name *SL*. Annotations can be added by writing `"`*text*`" at` *position* as always, but there are other ways. One alternative is to use, for example, `llabel(`*text, text, text, position, name*) as usual. However, this macro positions text by default with respect to `last []` which normally will be incorrect if breakers are automatically included with the element. In the latter case, enter the element body name as the fifth argument of `llabel()`. For example, `B: sl_ttbox` creates an element of which the invisible centre line has name *B* and the body has name *Body*, and can be labelled like a normal two-terminal element. If, however, breakers are included using `B: tt_box(,,,C,C)` then write, for example, `llabel(,Box 15,,,Body)` to place the label correctly.

11.2 One-terminal and composite SLD elements

The one-terminal elements have two components: a stem with optional breaker or slash symbol, and a head. SLD generators are shown in Figure 76, other one-terminal elements in Figure 77.

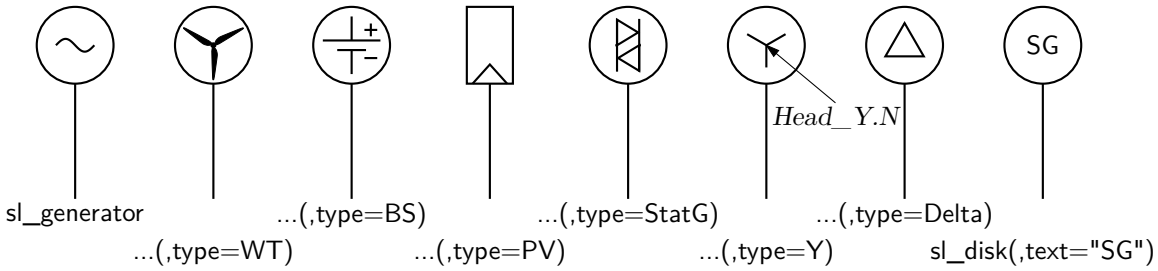


Figure 76: SLD generators, drawing direction `up_`.

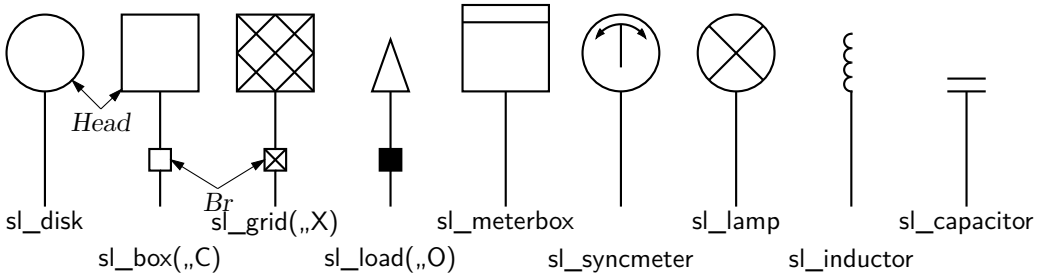


Figure 77: SLD one-terminal elements, drawing direction `up_`.

There are three arguments, as follows. The first argument is a linespec which defines the location and drawing direction of the element stem. The second argument is a sequence of semicolon-separated key-value pairs as necessary to customize and name the element head, of which the default name is *Head*. The third argument controls the presence and type of the object in the stem as for the two-terminal element breakers. The default breaker name is *Br* and the default slash name is *SL*, and the separation from the head is specified using global `sl_breakersep_` or the local `sep=`*expr* parameters as for the two-terminal elements.

A stem of zero length is allowed when only the element head is needed. Because a line segment of zero length has undefined direction, the first argument must be one of `U, D, L, R` (for up, down, left, right) or a number to set the direction in degrees, optionally followed by `at` *position* to set the position (`Here` by default). For example, `sl_box(45 at Here+(1,0))`.

The macros `sl_busbar(`*linespec, np, keys*`)` and `sl_ct(`*keys*`),` shown in Figure 78, are composite; that is, they are `[ ]` blocks with defined internal positions. For `sl_busbar`, these are *Start, End,* and *P1, P2, ... Pnp* where *np* is the value of the second argument.

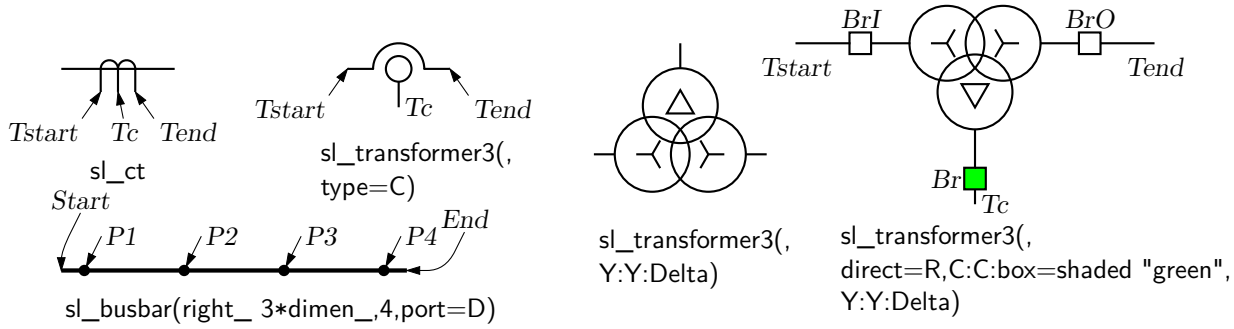


Figure 78: The `sl_busbar()` and some transformer variants.

For example, the line

```
line right_ 3cm__; sl_busbar(up_ 4.5cm__,5) with .P3 at Here
```

draws a vertical busbar at the end of a horizontal line.

12 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 the size of typeset text is independent of the pic language except when svg is being produced, and that line widths are independent of the scaling of drawn objects.

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

12.1 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
T: [linewid *= 1.5; up_; Q: bi_tr] with .Q.B at Here
  ground(at T.Q.E)
  resistor(up_ dimen_ from T.Q.C)
```

connects two resistors and a ground to an enlarged transistor. 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 body sizes for all circuit elements.

For more localized resizing, use, for example,

`pushdef(‘dimen_’,`*expression*`)` *drawing commands* `popdef(‘dimen_’)`

(but ensure that the drawing commands have no net effect on the `dimen_` stack).

For logic elements, the equivalent to the `dimen_` macro is `L_unit`, which has default value `(linewid/10)`.

The macros `capacitor, inductor,` and `resistor` have arguments that allow the body sizes to be adjusted individually. The macro `resized` mentioned previously can also be used.

12.2 Pic scaling

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

1. When generating final output after reading the `.PE` line, pic processors divide distances and sizes by the value of the environmental parameter `scale`, which is 1 by default. Therefore, the effect of assigning a value to `scale` at the beginning of the diagram is to change the drawing unit (initially 1 inch) throughout the figure. For example, the file `quick.m4` can be modified to use millimetres as follows:

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

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

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

of `scale` (gpic multiplies them all). Therefore, objects drawn to default sizes are unaffected by changing `scale` at the beginning of the diagram. To change default sizes, redefine the appropriate parameters explicitly.

2. Dpic implements a `scaled` attribute for objects, so you can enclose the entire diagram (or part of it) in `[ ]` brackets, thus: `[` ... *drawing commands* `] scaled` x where x is a scale factor.

3. The `.PS` line can be used to scale the entire drawing, regardless of its interior. Thus, for example, the line `.PS 100/25.4` scales the entire drawing to a width of 100 mm. Line thickness, text size, and dpic arrowheads are unaffected by this scaling.

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

4. The finished size of typeset text is independent of pic variables, but can be determined as in Section 14. Then, `"text" wid` x `ht` y tells pic the size of `text`, once the printed width x and height y have been found.

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

13 Writing macros

The m4 language is quite simple and is described in numerous documents such as the original reference [10] or in later manuals [17]. If a new circuit or other element is required, then it may suffice to modify and rename one of the library definitions or simply add an option to it. Hints for drawing general two-terminal elements are given in `libcct.m4`. However, if an element or block 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. If you develop a library of custom macros in the installation directory then the statement `include(`*mylibrary*`.m4)` can bring its definitions into play.

It may not be necessary to define your own macro if all that is needed is a small addition to an existing element that is defined in an enclosing `[ ]` block. After the element arguments are expanded, one argument beyond the normal list is automatically expanded before exiting the block, as mentioned near the beginning of Section 6. This extra argument can be used to embellish the element.

A macro is defined using quoted name and replacement text as follows:
`define(‘`*name*`’,‘`*replacement text*`’)`

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

Example 1: Custom two-terminal elements can often be defined by writing a wrapper for an existing element. For example, an enclosed thermal switch can be defined as shown in Figure 79.

```
define(‘thermalsw’,
 ‘dswitch(‘$1’,‘$2’,WDdBTh)
  {circle rad distance(last [].T,last line.c) at last line.c}’)
```

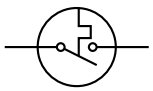


Figure 79: A custom thermal switch defined from the `dswitch` macro.

Example 2: In the following, two macros are defined to simplify the repeated drawing of a series resistor and series inductor, and the macro `tsection` defines a subcircuit that is replicated several times to generate Figure 80.

```
.PS
# ‘Tline.m4’
cct_init
hgt = elen_*1.5
ewd = dimen_*0.9
define(‘sresistor’,‘resistor(right_ ewd); llabel(,r)’)
define(‘sinductor’,‘inductor(right_ ewd,W); llabel(,L)’)
define(‘tsection’,‘sinductor
  { dot; line down_ hgt*0.25; dot
    parallel_(‘resistor(down_ hgt*0.5); rlabel(,R)’,
              ‘capacitor(down_ hgt*0.5); rlabel(,C)’)
    dot; line down_ hgt*0.25; dot }
  sresistor ’)

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

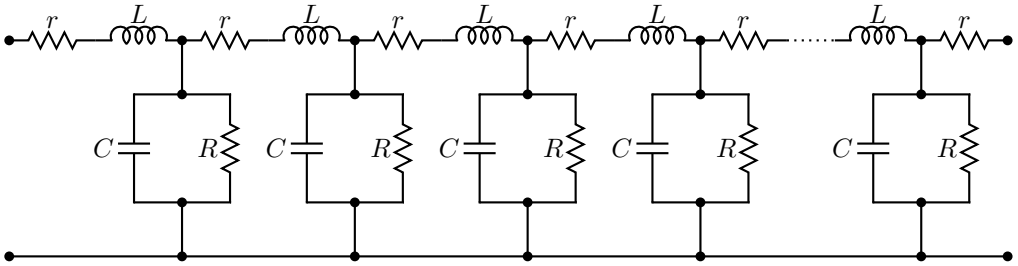


Figure 80: A lumped model of a transmission line, illustrating the use of custom macros.

Example 3: Figure 81 shows an element that is composed of several basic elements and that can be drawn at any angle. The labels always appear in their natural horizontal orientation.

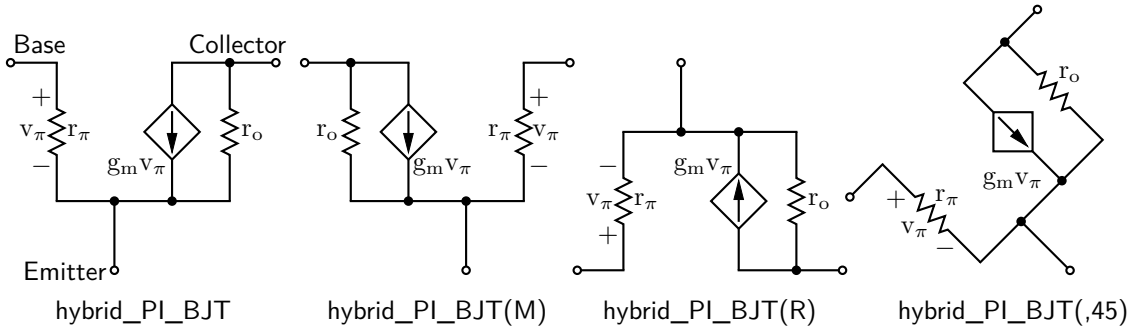


Figure 81: A composite element containing several basic elements

Two flags in the argument determine the circuit orientation with respect to the current drawing direction and whether a mirrored circuit is drawn. The key to writing such a macro is to observe that the pic language allows two-terminal elements to change the current drawing direction, so if arbitrary directions are allowed, the value of `rp_ang` should be saved and restored as necessary after each internal two-terminal element has been drawn; this can be avoided if only one drawing direction (`right_`, say) is required. A draft of such a macro for the general case follows:

```
#                                ‘hybrid_PI_BJT( [R][M], U|D|L|R|degrees )
#                                 drawn in current (or arg2) direction
#                                 R=right orientation; M=left-right mirror’
define(‘hybrid_PI_BJT’,
‘[ setdir_(‘$2’); hp_ang = rp_ang
                              # Size (and orientation) parameters:
   hunit = ifinstr(‘$1’,M,-)dimen_
   vunit = ifinstr(‘$1’,R,-)dimen_*3/2

    dot; {line to rvec_(0,-vunit/2)
  Emitter: dot(,,1)}
    {R_pi: resistor(from rvec_(-hunit*5/8,0) to rvec_(-hunit*5/8,vunit))
     point_(hp_ang); line to rvec_(-hunit/2,0) chop -lthick/2 chop 0
  Base: dot(,,1)}
    {dot(at rvec_(hunit*5/4,vunit)); R_o: resistor(to rvec_(0,-vunit))
     point_(hp_ang); line to R_pi.start chop -lthick/2}
    {dot(at rvec_(hunit*5/8,0)); CS: consource(to rvec_(0,vunit),I,R)
     point_(hp_ang); line to rvec_(hunit*9/8,0) chop -lthick/2 chop 0
  Collector: dot(,,1)}
#                                # Labels
  ‘"$\mathrm{r_\pi}$"’ at R_pi.c+vec_(hunit/4,0)
  ‘"$ + $"’ at R_pi.c+vec_(-hunit/6, vunit/4)
  ‘"$ - $"’ at R_pi.c+vec_(-hunit/6,-vunit/4)
  ‘"$\mathrm{v_\pi}$"’ at R_pi.c+vec_(-hunit/4,0)
  ‘"$\mathrm{g_m}$$\mathrm{v_\pi}$"’ at CS.c+vec_(-hunit*3/8,-vunit/4)
  ‘"$\mathrm{r_o}$"’ at R_o.c+vec_(hunit/4,0)
 ‘$3’ ] ’)
```

Example 4: A number of elements have arguments meant explicitly for customization. Figure 82 customizes the `source` macro to show a cycle of a horizontal sinusoid with adjustable phase given by argument 2 in degrees, as might be wanted for a 3-phase circuit:

```
define(‘phsource’,‘source($1,
#‘Set angle to 0, draw sinusoid, restore angle’
 m4smp_ang = rp_ang; rp_ang = 0
 sinusoid(m4h/2,twopi_/(m4h),
  ifelse(‘$2’,,,‘($2)/360*twopi_+’)pi_/2,-m4h/2,m4h/2) with .Origin at Here
 rp_ang = m4smp_ang,
$3,$4,$5)’)
```

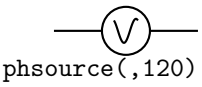


Figure 82: A source element customized using its second argument.

Example 5: Repeated subcircuits might appear only as the subcircuit and its mirror image, for example, so the power of the `vec_()` and `rvec_()` macros is not required. Suppose that an optoisolator is to be drawn with left-right or right-left orientation as shown in Figure 83.

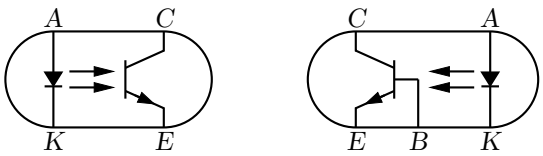


Figure 83: Showing `opto` and `opto(BR)` with defined labels.

The macro interface could be something like the following:
`opto( [L|R][A|B] ),`

where an R in the argument string signifies a right-left (mirrored) orientation and the element is of either A or B type; that is, there are two related elements that might be drawn in either orientation, for a total of four possibilities. Those who find such an interface to be too cryptic might prefer to invoke the macro as

opto(orientation=Rightleft;type=B),

which includes semantic sugar surrounding the R and B characters for readability; this usage is made possible by testing the argument string using the ifinstr() macro rather than requiring an exact match. A draft of the macro follows, and the file Optoiso.m4 in the examples directory adds a third type option.

```
#                           ‘opto([R|L][A|B])’
define(‘opto’,‘[{u = dimen_/2
 Q: bi_trans(up u*2,ifinstr(‘$1’,R,R),ifinstr(‘$1’,B,B)CBUdE)
 E: Q.E; C: Q.C; A:ifinstr(‘$1’,R,Q.e+(u*3/2,u),Q.w+(-u*3/2,u)); K: A-(0,u*2)
   ifinstr(‘$1’,B,line from Q.B to (Q.B,E); B: Here)
 D: diode(from A to K)
   arrow from D.c+(0,u/6) to Q.ifinstr(‘$1’,R,e,w)+(0,u/6) chop u/3 chop u/4
   arrow from last arrow.start-(0,u/3) to last arrow.end-(0,u/3)
 Enc: box rad u wid abs(C.x-A.x)+u*2 ht u*2 with .c at 0.5 between C and K
  ‘$2’ }]’)
```

Two instances of this subcircuit are drawn and placed by the following code, with the result shown in Figure 83.

```
Q1: opto
Q2: opto(type=B;orientation=Rightleft) with .w at Q1.e+(dimen_,0)
```

13.1 Macro arguments

Macro parameters are defined by entering them into specific arguments, and if an argument is blank then a default parameter is used. For the resistor macro, for example:

resistor(*linespec, cycles, chars, cycle wid*)

an integer (3, say) in the second argument specifies the number of cycles.

Recently for some macros, a mixed style has been adopted by which parameters can entered using keys. The previous case becomes

resistor(*linespec,* cycles=3;)

and the allowable keys for resistor are given in the macro definition, in this case on page 89.

Two macros assist this process. The first is

pushkey_(*string, key, default value,* [N])

so that in a macro, the line

pushkey_(‘$2’, wdth, dimen_*2)

checks macro argument 2 for the substring wdth=*expression*. If found, the macro m4wdth is defined, using pushdef, to equal (*expression*) with enclosing parentheses omitted if the fourth argument of pushkey_ is nonblank as would be required if m4wdth were to be non-numeric. If the substring wdth= is not found, then m4wdth is given the default value (dimen_*2). The key wdth normally should not be a macro name.

In addition, the macro

pushkeys_(*string, keysequence*)

applies pushkey_() to each of the terms of its *keysequence* (second) argument. Each term of the semicolon-separated second argument sequence consists of the rightmost three arguments of pushkey_ separated by colons (:) rather than commas. Normal good practice cancels pushed key definitions at macro exit, for example, popdef(‘m4*string1*’,‘m4*string2*’, ...).

The macros setkey_() and setkeys_() are similar to pushkey_() and pushkeys_() respectively but use the m4 define command rather than pushdef.

For example, consider the elementary example of a custom box macro:

```
define(‘custombox’,
‘pushkeys_(‘$1’,wdth:boxwid:; hght:boxht:; name::N; text::N)
 ifelse(m4name,,,m4name:) box wid m4wdth ht m4hght ifelse(m4text,,,"m4text")
 popdef(‘m4wdth’,‘m4hght’,‘m4name’,‘m4text’)’)
```

Then `custombox(wdth=2; name=B1; text=Hello)` first causes the macros `m4wdth`, `m4hght`, `m4name`, and `m4text` to be created with values `(2)`, `(boxht)`, `B1`, and `Hello` respectively, and `custombox` evaluates to

`B1: box wid (2) ht (boxht) "Hello"`.

As another example, the macro `sarrow(`*linespec*`,` *keys*`)` can generate the custom arrows shown below the three native arrows in Figure 84. The defined keys are `type=`; `wdth=`; `lgth=`; `shaft=`; `head=`; `hook=`; and `name=`. Many variations of these arrowheads could be created by adding keys.

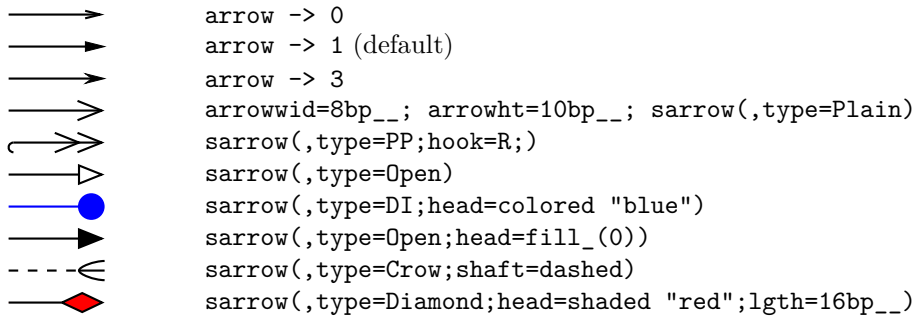


Figure 84: The three dpic native arrows and others generated by `sarrow(`*linespec*`,` *keys*`)`.

14 Interaction with LaTeX

The sizes of typeset labels and other TeX boxes are generally unknown prior to processing the diagram by LaTeX. Although they are not needed for many circuit diagrams, these sizes may be required explicitly for calculations or implicitly for determining the diagram bounding box. The following example shows how text sizes can affect the overall size of a 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 size of the text to the left and right of the box, and the diagram is generated using default text size values. One solution to this problem 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.

Often, a better solution is to process the diagram twice. The diagram source is processed as usual by m4 and a pic processor, and the main document source is LaTeXed to input the diagram and format the text, and also to write the text dimensions into a supplementary file. Then the diagram source is processed again, reading the required dimensions from the supplementary file and producing a diagram ready for final LaTeXing. This hackery is summarized below, with an example in Figure 85.

- 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, or alternatively, invoke the macros `\boxdims` and `boxdim` described later. The argument of `s_box` need not be text exclusively; it can be anything that produces a TEX box, for example, `\includegraphics`.

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



Figure 85: Macro `s_box` sets string dimensions automatically when processed twice. If two or more arguments are given to `s_box`, they are passed through `sprintf`. The bounding box is shown.

The macro `s_box(`*text*`)` evaluates initially to

`"\boxdims{`*name*`}{`*text*`}" wid boxdim(`*name*`,w) ht boxdim(`*name*`,v)`

On the second pass, this is equivalent to

"*text*" `wid` *x* `ht` *y*

where *x* and *y* are the typeset dimensions of the LATEX input text. If `s_box` is given two or more arguments as in Figure 85 then they are processed by `sprintf`.

The argument of `s_init`, which should be unique within *jobname*`.dim`, is used to generate a unique `\boxdims` first argument for each invocation of `s_box` in the current file. If `s_init` has been omitted, the symbols "**!!**" are inserted into the text as a warning. Be sure to quote any commas in the arguments. Since the first argument of `s_box` is LATEX source, make a rule of quoting it to avoid comma and name-clash problems. For convenience, the macros `s_ht`, `s_wd`, and `s_dp` evaluate to the dimensions of the most recent `s_box` string or to the dimensions of their argument names, if present.

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

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

has the same effect as

`box "\Huge Hi there!"`

except that the line

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

is written into file *jobname*`.dim` (and the numerical values depend on the current font). These definitions are required by the `boxdim` macro described below.

The LATEX macro

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

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

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

`m4` *library files dimension file diagram source file* `...`

Here is a second small example. Suppose that the file `tsbox.m4` contains the following:

```
\documentclass{article}
\usepackage{boxdims,ifpstricks(pstricks,tikz)}
\begin{document}
.PS
cct_init s_init(unique) sinclude(tsbox.dim)
[ source(up_,AC); llabel(,s_box(AC supply)) ]; showbox_
```

```
.PE
\end{document}
```

The file is processed twice as follows:

```
m4 pgf.m4 tsbox.m4 | dpic -g > tsbox.tex; pdflatex tsbox
m4 pgf.m4 tsbox.m4 | dpic -g > tsbox.tex; pdflatex tsbox
```

The first command line produces a file `tsbox.pdf` with incorrect bounding box. The second command reads the data in `tsbox.dim` to size the label correctly. The equivalent pstricks commands (note the `ifpstricks` macro in the second line of the diagram source) are

```
m4 pstricks.m4 tsbox.m4 | dpic -p > tsbox.tex; latex tsbox
m4 pstricks.m4 tsbox.m4 | dpic -p > tsbox.tex; latex tsbox; dvips tsbox
```

Objects can be taylored to their attached text by invoking `\boxdims` and `boxdim` explicitly. The small source file in Figure 86, for example, produces the box in the figure.

```
.PS
# 'eboxdims.m4'
sinclude(Circuit_macros.dim)  # The input file is Circuit_macros.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
```

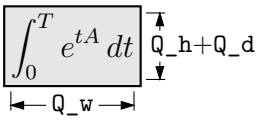


Figure 86: Fitting a box to typeset text.

The figure is processed twice, as described previously. The line `sinclude(`*jobname*`.dim)` reads the named file if it exists. The macro `boxdim(`*name,suffix,default*`)` from `libgen.m4` expands the expression `boxdim(Q,w)` to the value of `Q_w` if it is defined, else to its third argument if defined, else to 0, the latter two cases applying if *jobname*`.dim` doesn't exist yet. The values of `boxdim(Q,h)` and `boxdim(Q,d)` are similarly defined and, for convenience, `boxdim(Q,v)` evaluates to the sum of these. Macro `pt__` is defined as `*scale/72.27` in `libgen.m4`, to convert points to drawing coordinates.

Sometimes a label needs a plain background in order to blank out previously drawn components overlapped by the label, as shown on the left of Figure 87.



Figure 87: Illustrating the `f_box` macro.

The technique illustrated in Figure 86 is automated by the macro `f_box(`*boxspecs*, *label arguments*`)`. For the special case of only one argument, e.g., `f_box(Wood chips)`, this macro simply overwrites the label on a white box of identical size. Otherwise, the first argument specifies the box characteristics (except for size), and the macro evaluates to

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

For example, the result of the following command is shown on the right of Figure 87.

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

More tricks can be played. The example

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

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

15 PSTricks and other tricks

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

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

Some commonly required effects are particularly simple. For example, the rotation of text by PSTricks postprocessing is illustrated by the file

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

which contains both horizontal text and text rotated 90° along the vertical line. This rotation of text is also implemented by the macro `rs_box([angle=`*degrees*`;] `*text*`[,`*expr, expr ...*`])`, which is similar to `s_box` but rotates its argument by 90°, a default angle that can be changed by preceding invocation with `define(‘text_ang’,`*degrees*`)` or by starting the first argument with `angle=`*degrees*`;` where *degrees* is a decimal number (not an expression). The `rs_box` macro requires either PSTricks or Ti*k*z PGF and, like `s_box`, it calculates the size of the resulting text box but requires the diagram to be processed twice.

The macro `r_text(`*degrees, text,* `at` *position*`)` works under PSTricks, Ti*k*z PGF, and SVG, the last requiring processing twice. The *degrees* argument is a decimal constant (not an expression) and the text is a simple string without quotes. The text box is not calculated.

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

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

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

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

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

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

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

15.1 Ti*k*z with pic

Arbitrary pic output can be inserted into a `\tikzpicture` environment. The trick is to keep the pic and Ti*k*z coordinate systems the same. The lines

```
\begin{tikzpicture}[scale=2.54]
\end{tikzpicture}%
```

in the `dpic -g` output must be changed to

```
\begin{scope}[scale=2.54]
\end{scope}%
```

This is accomplished, for example, by adapting the `\mtotex` macro of Section 2.1.4 as follows:

```
\newcommand\mtotikz[1]{\immediate\write18{m4 pgf.m4 #1.m4 | dpic -g
  | sed -e "/begin{tikzpicture}/s/tikzpicture/scope/"
```

```
-e "/end{tikzpicture}/s/tikzpicture/scope/" > #1.tex}\input{./#1.tex}}%
```

Then, from within a Ti*k*z pictdure, `\mtotikz{`*filename*`}` will create *filename*`.tex` from *filename*`.m4` and read the result into the Ti*k*z code.

In addition, the Ti*k*z code may need to refer to nodes defined in the pic diagram. The included m4 macro `tikznode(`*tikz node name,[position],[string]*`)` defines a zero-size Ti*k*z node at the given pic position, which is `Here` by default. This macro must be invoked in the outermost scope of a pic diagram, and the `.PS` *value* scaling construct may not be used.

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

The issues related to web publishing are similar to those for other documents containing both graphics and text. Here the important factor is that gpic `-t` generates output containing tpic `\special` commands, which must be converted to the desired output, whereas dpic can generate several alternative formats, as shown in Figure 88. One of the easiest methods for producing web documents is to generate postscript as usual and to convert the result to pdf format with Adobe Distiller or equivalent.

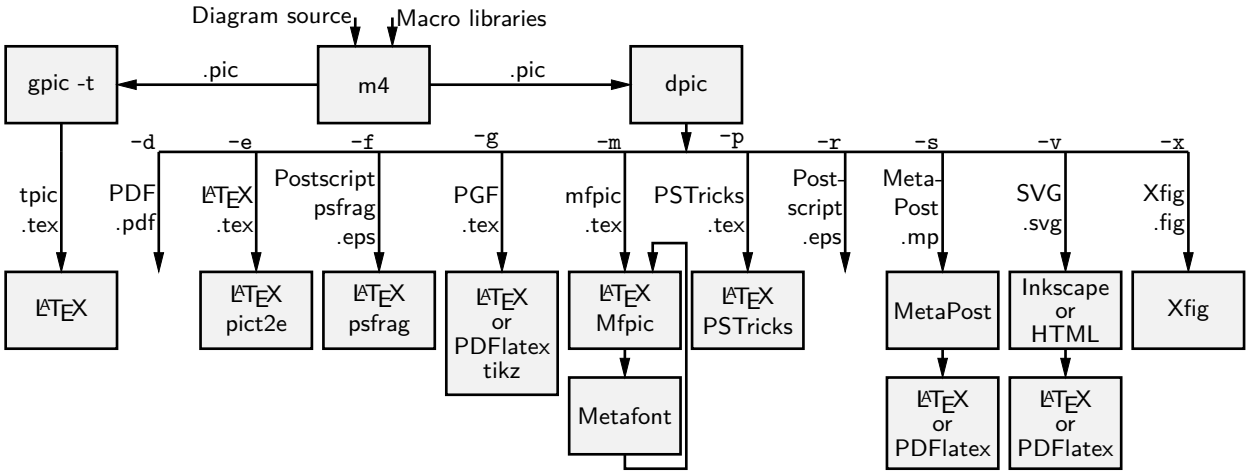


Figure 88: Output formats produced by gpic`-t` and dpic. SVG output can be read by Inkscape or used directly in web documents.

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

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

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

Then append one or more diagrams by using the equivalent of

`m4 <`*installdir*`>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 14. 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 <`*installdir*`>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}` as usual.

The dpic processor can generate other output formats, as illustrated in Figure 88 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.

17 Developer's notes

In the course of writing a book in the late 1980s when there was little available for creating line diagrams in LaTeX, I wished to eliminate the tedious coordinate calculations required by the LaTeX picture objects that I was then using. The pic language seemed to be a good fit for this purpose, and I took a few days off to write a pic-like interpreter (dpic). The macros in this distribution and the interpreter are the result of that effort, drawings I have had to produce since, and suggestions received from others. The emphasis throughout has been to produce a few types of diagrams well rather than attempting to satisfy the needs of everyone.

Dpic has been upgraded over time to generate mfpic, MetaPost [6, 19], raw Postscript, Postscript with `psfrag` tags, raw PDF, PSTricks, and Ti*k*Z PGF output, the latter two my preference because of their quality and flexibility, including facilities for colour and rotations, together with simple font selection. Xfig-compatible output was introduced early on to allow the creation of diagrams both by programming and by interactive graphics. SVG output was added relatively recently, and seems suitable for producing web diagrams directly and for further editing by the Inkscape interactive graphics editor. The latest addition is raw PDF output, which has very basic text capability and is most suitable for creating diagrams without labels.

The simple pic language is but one of many available tools for creating line graphics. Consequently, the main value of this distribution is not necessarily in the use of a specific language but in the element data encoded in the macros, which have been developed with reference to standards and refined over decades, and which now total thousands of lines. The learning curve of pic compares well with other possibilities but some of the macros have become less readable as more options and flexibility have been added, and if starting over today, perhaps I would change some details. Compromises have been made to preserve the compatability of some of the older macros and also to retain reasonable consistency over the various postprocessors. No choice of tool is without compromise, and producing good graphics seems to be time consuming, no matter how it is done, but the payoff can be worth the effort.

Instead of using pic macros, I preferred the equally simple but more powerful m4 macro processor, and therefore m4 is required here, although dpic now supports pic macros. Free versions of m4 are available for Unix and its descendents, Windows, and other operating systems. Additionally, the simplicity of m4 and pic enables the writing of custom macros, which are mentioned from time to time in this manual and included in some of the examples.

If starting over today would I not just use one of the other drawing packages available these days? It would depend on the context, but pic remains a good choice for line drawings because it is easy to learn and read but powerful enough (that is, Turing-complete) for coding the geometrical calculations required for precise component sizing and placement. It would be nice if arbitrary rotations and scaling were simpler, if a general path element with clipping were available as in Postscript, and if adding color across postprocessors were easier. However, all the power of Postscript or Ti*k*z PGF, for example, remains available, as arbitrary postprocessor code can be included with pic code.

The dpic interpreter has several output-format options that may be useful. The `eepicemu` and `pict2e` extensions of the primitive LaTeX picture objects are supported. The mfpic output allows the

production of Metafont alphabets of circuit elements or other graphics, thereby essentially removing dependence on device drivers, but with the complication of treating every alphabetic component as a TEX box. The xfig output allows elements to be precisely defined with dpic and interactively placed with xfig. Similarly, the SVG output can be read directly by the Inkscape graphics editor, but SVG can also be used directly for web pages. Dpic will also generate low-level MetaPost or Postscript code, so that diagrams defined using pic can be manipulated and combined with others. I learned to great benefit that the Postscript output can be imported into CorelDraw and Adobe Illustrator for further processing, so that detailed diagram components produced by pic can be combined with effects best acheived using a wysiwyg drawing program. With raw Postscript, PDF, and SVG output however, the user is responsible for ensuring that the correct fonts are provided and for formatting the text.

Could more be done in other contexts? Of course; consider the many symbols in ISO 7000, for example. Application-related libraries could be created in other fields (but I preferred to stay with applications about which I knew something).

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

18 Bugs

This section provides hints and a list of common errors.

First of all, be aware that old versions of LATEX, dpic, and these macros are not always compatible. Updating an installation to current versions is often the way to eliminate mysterious error messages. These macros use features available in GNU M4 (1.4.19) or later.

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.

Maintaining reasonable compatibility with both gpic and dpic and, especially, with different post-processors, has resulted in some macros becoming more complicated than is preferable. Furthermore, some of the newer macros make use of dpic facilities not available with gpic.

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

1. **Misconfiguration:** One of the configuration files listed in Section 2.2 and `libgen.m4` *must* be read by m4 before any other library macros. Otherwise, the macros assume default configuration. To aid in detecting the default condition, a `WARNING` comment line is inserted into the `pic` output. If only PSTricks is to be used, for example, then the simplest strategy is to set it as the default processor by typing "make psdefault" in the installation directory to change the mention of `gpic` to `pstricks` near the top of `libgen.m4`. Similarly if only Ti*k*z PGF will be used, change `gpic` to `pgf` using the Makefile. The package default is to read `gpic.m4` for historical compatibility. The processor options must be chosen correspondingly, `gpic -t` for `gpic.m4` and, most often, `dpic -p` or `dpic -g` when dpic is employed. For example, the pipeline for PSTricks output from file `quick.m4` is

   `m4 -I` *installdir* `pstricks.m4 quick.m4 | dpic -p > quick.tex`

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

   `m4 -I` *installdir* `pgf.m4 quick.m4 | dpic -g > quick.tex`

   Any non-default configuration file must appear explicitly in the command line or in an `include()` statement.

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

   ```
   line from A to B; resistor from B to C; ground at D
   ```

   when it should be

   ```
   line from A to B; resistor(from B to C); ground(at D)
   ```

This error is caused by an unfortunate inconsistency between pic object attributes and the way m4 and pic pass macro arguments.

3. **Commas:** Macro arguments are separated by commas, so any comma that is part of an argument must be protected by parentheses or quotes. Thus,

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

   produces an error, whereas

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

   do not. The parentheses are preferred. For example, a macro invoked by circuit elements contained the line

   `command "\pscustom[fillstyle=solid‘,’fillcolor=m4fillv]{%"`

   which includes a comma, duly quoted. However, if such an element is an argument of another macro, the quotes are removed and the comma causes obscure “too many arguments” error messages. Changing this line to

   `command sprintf("\pscustom[fillstyle=solid,fillcolor=m4fillv]{%%")`

   cured the problem because the protecting parentheses are not stripped away.

   As a second example, the expansion of `rgbstring(`*red frac,green frac,blue frac*`)` for postprocessor PSTricks or pgf contains a comma, so this macro is fragile when part of an argument of another macro. One cure is to replace `rgbstring(...)` in the problematic argument by a name, `newcolor` say, and define a pic macro: `define newcolor {rgbstring(...)}` so that `newcolor` gets replaced by the color specification when needed during dpic execution.

4. **Default directions and lengths:** The *linespec* argument of element macros defines a straight-line segment, which requires the equivalent of four parameters to be specified uniquely. If information is omitted, default values are used. Writing

   `source(up_)`

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

   `source(0.5)`

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

   `source(up_ elen_).`

   The effect of a *linespec* argument is independent of any direction set using the `Point_` or similar macros. To draw an element at an obtuse angle (see Section 7) try, for example,

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

5. **Mixing m4 and dpic code:** It is easy to forget that m4 finishes before pic processing begins. Consequently, it may be puzzling that the following mix of a pic loop and the m4 macro `s_box` does not appear to produce the required result:

   `for i=1 to 5 do {s_box(A[i]); move }`

   In this example, the `s_box` macro is expanded only once and the index `i` is not a number. This particular example can be repaired by using an m4 loop:

   `for_(1,5,1,‘s_box(A[m4x]); move’)`

   Note that the loop index variable `m4x` is automatically defined.

   Another potential problem is that macros like `vec_()` and `rvec_()` attempt to produce the simplest output possible, depending on their arguments, in order to facilitate debugging. This is accomplished by checking for multiplication by 0, 1, or -1 and simplifying accordingly, and also checking for addition or subtraction of 0. However, their arguments may change

inside a pic loop, for example, causing difficulties. A recent switch in `libgen.m4` (that is, `define(‘robustcode_’,1)`), will avoid these changes, if you must, for robustness at the expense of longer output expressions. The robust macros `vec_r()` and `rvec_r()` are also provided. Better strategies may be to avoid argument-sensitive m4 macros in pic loops or to use m4 loops instead.

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

   `"‘‘inverse’’"`

   will become

   `"‘inverse’"`.

   The cure is to add single quotes in pairs as necessary.

   If text containing single quote characters causes difficulties then replace the LaTeX single quote by `\char39` or disable the m4 quote characters temporarily as shown:

   `changequote(,)` *text containing single quotes* `changequote(‘,)`

   The only subtlety required in writing m4 macros is deciding when to quote macro arguments. In the context of circuits it seemed best to assume that arguments would not be protected by quotes at the level of macro invocation, but should be quoted inside each macro. There may be cases where this rule is not optimal or where the quotes could be omitted, and there are rare exceptions such as the `parallel_` macro.

   To keep track of paired single quotes, parentheses “`(, )`,” braces “`{, }`,” and brackets “`[, ]`,” use an editor that highlights these pairs. For example, the vim editor highlights single quotes with the command `:set mps+=‘:’`.

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

   `"$0$" rjust below`

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

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

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

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

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

   Library macros intended only for internal use have names that begin with `m4` or `M4` to avoid name clashes, but in addition, a good rule is to quote all LaTeX in the diagram input.

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

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

   with the LaTeX macro

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

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

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

rather than

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

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

10. **Position of elements that are not 2-terminal:** The *linespec* argument of elements defined in `[ ]` blocks must be understood as defining a direction and length, but not the position of the resulting block. In the pic language, objects inside these brackets are placed by default *as if the block were a box*. Place the element by its compass corners or defined interior points as described in the first paragraph of Section 6 on page 20, for example

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

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

12. **Line continuation:** A line is continued to the next if the rightmost character is a backslash or, with dpic, if the backslash is followed immediately by the `#` character. A blank after the backslash, for example, produces a pic error.

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

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

15. **m4 `-I` error:** Some old versions of m4 may not implement the `-I` option or the `M4PATH` environment variable that simplify file inclusion. The simplest course of action is probably to install GNU m4, which is free and widely available. Otherwise, all `include(`*filename*`)` statements in the libraries and calling commands have to be given absolute *filename* paths. You can define the `HOMELIB_` macro in `libgen.m4` to the path of the installation directory and change the library include statements to the form `include(HOMELIB_''`*filename*`)`.

19 List of macros

The following table lists macros in the libraries, configuration files, and selected macros from example diagrams. Some of the sources in the `examples` directory contain additional macros, such as for flowcharts, Boolean logic, ICs, connectors, and binary trees.

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

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

A B C D E F G H I J K L M N O P R S T U V W X Y Z

A

`above_`	gen	string position above relative to current direction
`abs_(`*number*`)`	gen	absolute value function

`ACsymbol(at` *position, len, ht,* `[n:][A]U|D|L|R|`*degrees*`)`
cct — draw a stack of n (default 1) AC symbols (1-cycle sine waves); If arg 4 contains A, two arcs are used. The current drawing direction is default, otherwise Up, Down, Left, Right, or at *degrees* slant; (Section 4.2) e.g., `ebox; {ACsymbol(at last [],,dimen_/8)}`

`adc(`*width, height, nIn, nN, nOut, nS*`)`
cct — Analog-digital converter with defined width, height, and number of inputs `In`i, top terminals `N`i, ouputs `Out`i, and bottom terminals `S`i

`addtaps([`*arrowhd*`| type=`*arrowhd*`;name=`*Name*`],` *fraction, length, fraction, length,* $\cdots$`)`
cct — Add taps to the previous two-terminal element. *arrowhd* is blank or one of `. - <- -> <->`. Each fraction determines the position along the element body of the tap. A negative length draws the tap to the right of the current direction; positive length to the left. Tap names are Tap1, Tap2, $\cdots$ by default or Name1, Name2, $\cdots$ if specified (Section 6)

`adjust([at` *position*`],` *keys*`)` cct — Ajdustment screwhead in a [ ] block. *keys:* `size=`*expression;* `angle=`*degrees;* `slotwid=`*expression;* `circle=`*attributes;*

`along_(`*linear object name*`)` gen — short for `between` *name*`.start and` *name*`.end`

`Along_(`*LinearObj,distance,*`[R])`
gen — Position arg2 (default all the way) along a linear object from `.start` to `.end` (from `.end` to `.start` if arg3=`R`)

`amp(`*linespec*`,` *size*`,` *attributes*`)` cct — amplifier (Section 4.2)

`And, Or, Not, Nand, Nor, Xor, Nxor, Buffer`
log — Wrappers of `AND_gate`, ... for use in the `Autologix` macro

`AND_gate(`n`,` `[N][B],` `[`*wid*`,` `[`*ht*`]],` *attributes*`)`
log — 'and' gate, 2 or n inputs ($0 \le n \le 16$) drawn in the current direction; `N`: negated inputs; `B`: box shape. Alternatively, `AND_gate(`*chars*`,` `[B],` *wid*`,` *ht*`,` *attributes*`)`, where arg1 is a sequence of letters `P|N` to define normal or negated inputs. (Section 9)

`AND_gen(`n`,` *chars*`,` `[`*wid*`,` `[`*ht*`]],` *attributes*`)`
log — general AND gate: n=number of inputs ($0 \le n \le 16$); *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. Otherwise, arg1 can be a sequence of letters `P|N` to define normal or negated inputs; arg2 is as above except that `[N]I` is ignored. Arg 5 contains body attributes.

`AND_ht` log — height of basic 'and' and 'or' gates in `L_units`, default 6

`AND_wd` log — width of basic 'and' and 'or' gates in `L_units`, default 7

`antenna(at` *location*`, T, A|L|T|S|D|P|F, U|D|L|R|`*degrees*`)`

cct — antenna, without stem for nonblank 2nd arg; arg3 is
`A`: aerial (default),
`L`: loop,
`T`: triangle,
`S`: diamond,
`D`: dipole,
`P`: phased,
`F`: fork;
arg4 specifies Up, Down, Left, Right, or angle from horizontal (default 90) (Section 6)

`arca(`*absolute chord linespec*`, ccw|cw,` *radius*`,` *modifiers*`)`

gen — arc with acute angle (obtuse if radius is negative), drawn in a [ ] block

`ArcAngle(`*position, position, position, radius, modifiers, label*`)`

gen — Arc angle symbol drawn ccw at arg2. Arg4 is the radius from arg2; arg5 contains line attributes, e.g., `thick linethick/2 ->`; arg6 is an optional label at mid-arc

`arcd(`*center*`,` *radius*`,`*start degrees*`,`*end degrees*`)`

gen — Arc definition (see `arcr`), angles in degrees (Section 3.3)

`arcdimension_(`*arcspec*`,` *offset*`,` *label*`, D|H|W|`*blank width*`,` *tic offset*`, -> | <-)`

gen — arcs with arrowheads for dimensioning an angle in a technical drawing, similar to `dimension_`; Arg1 defines the attributes of an invisible arc: `arc invis` arg1.
Arg2 is the radial displacement (possibly negative) of the dimension arrows from the arc.
Arg3: label, normally a number or number with unit symbol.
Arg4: if arg3 is `s_box(...)` or `rs_box(...)` and arg4 is one of `D,H,W` then arg4 means:
`D:` blank width is the diagonal length of arg3;
`H:` blank width is the height of arg3 + `textoffset*2`;
`W:` blank width is the width of arg3 + `textoffset*2`;
otherwise arg4 is the absolute blank width.
Arg5 is `-> | <-` to designate a single arrowhead at the end or start of the reference arc; otherwise both arrowheads are drawn by default.

`arcr(`*center*`,`*radius*`,`*start angle*`,`*end angle*`,`*modifiers*`,`*ht*`)`

gen — Arc definition. If arg5 contains `<-` or `->` then a midpoint arrowhead of height equal to arg6 is added. Arg5 can contain modifiers (e.g. outlined "red"), for the arc and arrowhead. Modifiers following the macro affect the arc only, e.g., `arcr(A,r,0,pi_/2,->) dotted ->` (Section 3.3)

`arcto(`*position 1*`,` *position 2*`,` *radius*`, [dashed|dotted])`

gen — line toward position 1 with rounded corner toward position 2

`arcwinding(`*winding diam, start degrees, end degrees, nturns, core centre rad, core width,* `"`*core color*`")`

cct — winding drawn on an arc. The complete spline is drawn, then parts of it are overwritten with the background color (default white). Negative arg5 (default `dimen_` puts winding terminals at the outside.
Example: `W: arcwinding(1.2,-20,20,8, -1,0.8)`
(Section 6)

`array(`*variable, expr1, expr2, ...*`)`

dpictools — Populate a singly-subscripted array: *var*`[1]=`*expr1;* *var*`[2]=`*expr2;* `....`

`array2(`*variable, expr1, expr2, ...*`)`

dpictools — Populate a doubly-subscripted array: *var*`[`*expr1*`,1]=`*expr2;* *var*`[`*expr1*`,2]=`*expr3;* `....`

`arraymax(`*data array, n, index name, value*`)`

dpictools — Find the index in *array[1:n]* of the first occurrence of the maximum array element value. The value is assigned if arg4 is nonblank; example: `array(x,4,9,8,6);` `arraymax( x,4,i )` assigns 2 to *i*, and `arraymax( x,4,i,m )` assigns 2 to *i* and 9 to *m*.

`arraymin(`*data array, n, index name, value*`)`

dpictools — Find the index in *array[1:n]* of the first occurrence of the minimum array element value. The value is assigned if arg4 is nonblank; see `arraymax`.

`arrester(`*linespec*`,` *chars*`[D[L|R]],` *body len*`[:`*arrowhead ht*`],` *body ht*`[:`*arrowhead wid*`],` *attributes* `)`

cct — Arg2 *chars:*
`G=` spark gap (default)
`g=` general (dots)
`E=` gas discharge
`S=` box enclosure
`C=` carbon block
`A=` electrolytic cell
`H=` horn gap
`P=` protective gap
`s=` sphere gap
`F=` film element
`M=` multigap
Modifiers appended to arg2:
`R=` right orientation
`L=` left orientation
`D=` for `S, E` only, create a 3-terminal composite element with terminals *A*, *B*, *G*, placed as a block since Arg1 now determines length and direction but not position.
(Section 4.2)

`arrowline(`*linespec*`)` — cct — line (dotted, dashed permissible) with centred arrowhead (Section 4.2)

`assign3(`*name, name, name, arg4, arg5, arg6*`)`
gen — Assigns $1 = arg4 if $1 is nonblank; similarly $2 = arg5 and $3 = arg6

`AutoGate` — log — Draw the tree for a gate as in the `Autologix` macro. No inputs or external connections are drawn. The names of the internal gate inputs are stacked in `'AutoInNames'`

`Autologix(`*Boolean function sequence*`,[N[oconnect]][L[eftinputs]][R][V][M][;offset=`*value*`])`
log — Draw the Boolean expressions defined in function notation. The first argument is a semicolon (;)-separated sequence of Boolean function specifications using the functions `And, Or, Not, Buffer, Xor, Nand, Nor, Nxor` with variables, e.g., `Autologix(And(Or(x1,~x2),Or(~x1,x2)));`. Each function specification is of the form *function*`(`*arguments*`) [@`*attributes*`]`. Function outputs are aligned vertically but appending `@`*attributes* to a function can be used to place it; e.g., `Nand(~A,B) @with .n at last [].s+(0,-2bp__).` The function arguments are variable names or nested Boolean functions. Each unique variable *var* causes an input point `In`*var* to be defined. Preceding the variable by a `~` causes a NOT gate to be drawn at the input. The inputs are drawn in a row at the upper left by default. An `L` in arg2 draws the inputs in a column at the left; `R` reverses the order of the drawn inputs; `V` scans the expression from right to left when listing inputs; `M` draws the left-right mirror image of the diagram; and `N` draws only the function tree without the input array. The inputs are labelled `In1`, In2, ... and the function outputs are `Out1`, Out2, ... Each variable *var* corresponds also to one of the input array points with label `In`*var*. Setting `offset=`*value* displaces the drawn input list in order to disambiguate the input connections when `L` is used.
In the (possibly rare) case where one or more inputs of a normal function gate is to have a NOT-circle, an additional first argument of the function is inserted, of the form `[`*charseq*`]`, where *charseq* is a string containing the characters `P` for a normal input or `N` for a negated input, the length of the string equal to the number of gate inputs. Example:
`Autologix(Xor([PN],And(x,y),And(x,y)),LRV)`

B

`basename_(`*string sequence, separator*`)`
gen — Extract the rightmost name from a sequence of names separated by arg2 (default dot ".")

`battery(`*linespec*`,`*n*`,R)` — cct — n-cell battery: default 1 cell, R=reversed polarity (Section 4.2)

`b_` — gen — blue color value

`b_current(`*label*`, `*pos*`, In|Out, Start|End, `*frac*`)`
cct — labelled branch-current arrow to *frac* between branch end and body (Section 4.3)

`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 (Section 6)

`below_` gen — string position relative to current direction

`Between_(`*Pos1, Pos2,distance,*`[R])`
gen — Position *distance* from *Pos1* toward *Pos2*. If the fourth arg is `R` then from *Pos2* toward *Pos1*.

`binary_(`*n*`, [`*m*`])` gen — binary representation of *n*, left padded to *m* digits if the second argument is nonblank

`bisect(`*function name, left bound, right bound, tolerance, variable*`))`
dpictools — Solve *function*$(x) = 0$ by the method of bisection. Like `findroot` but uses recursion and is without a `[]` box. The calculated value is assigned to the variable named in the last argument (Section 2.2). Example:
```
define parabola { $2 = ($1)^2 - 1 };
bisect( parabola, 0, 2, 1e-8, x ).
```

`bi_trans(`*linespec*`,L|R,`*chars*`,E)`
cct — bipolar transistor, core left or right; chars:
`BU`: bulk line
`B`: base line and label
`S`: Schottky base hooks
`uEn|dEn`: emitters E0 to En
`uE|dE`: single emitter
`Cn|uCn|dCn`: collectors C0 to Cn; `u` or `d` add an arrow
`C`: single collector; `u` or `d` add an arrow
`G`: gate line and location
`H`: gate line;
`L`: L-gate line and location
`[d]D`: named parallel diode
`d`: dotted connection
`[u]T`: thyristor trigger line
arg 4 = `E`: envelope (Section 6.1)

`bi_tr(`*linespec*`,L|R,P,E)` cct — left or right, N- or P-type bipolar transistor, without or with envelope (Section 6.1)

`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 (Section 14)

`BOX_gate(`*inputs, output, swid, sht, label, attributes* `)`
log output=[P|N], inputs=[P|N]..., sizes swid and sht in `L_units` (default `AND_wd` = 7) (Section 9)

`bp__` gen big-point-size factor, in scaled inches, (`*scale/72`)

`bswitch(`*linespec*`, [L|R],chars)`
cct pushbutton switch R=right orientation (default L=left); chars: O= normally open, C=normally closed

`BUFFER_gate(`*linespec*`, [N|B],` *wid, ht,* `[N|P]*, [N|P]*, [N|P]*,` *attributes*`)`
log basic buffer, dfault 1 input or as a 2-terminal element, arg2: `N`: negated input, `B`: box gate; arg 5: normal (`P`) or negated `N`) inputs labeled In1 (Section 9)

`BUFFER_gen(`*chars,wd,ht*`,[N|P]*,[N|P]*,[N|P]*,` *attributes*`)`
log general buffer, *chars:*
`T`: triangle,
`[N]O`: output location `Out` (`NO` draws circle `N_Out`);
`[N]I, [N]N, [N]S, [N]NE, [N]SE` input locations;
`C`: centre location.
Args 4-6 allow alternative definitions of respective `In, NE,` and `SE` argument sequences

`BUF_ht` log basic buffer gate height in `L_units`, default 4

`BUF_wd` log basic buffer gate width in `L_units`, default 3.5

`buzzer( U|D|L|R|`*degrees, size,*`[C])`
cct buzzer, *In1* to *In3* defined, C=curved (Section 6)

C

`cangle(`*Start, End*`,[d])` gen Angle in radians of the sector at arg2 with arm ends given by arg1 and arg3 (degrees if arg4=d).

`capacitor(`*linespec*`,`*chars*`,R,` *height, wid*`)`
cct capacitor, *chars*:
`F` or blank: flat plate
`dF` flat plate with hatched fill
`C` curved-plate
`dC` curved-plate with variability arrowhead
`CP` constant phase element
`E` polarized boxed plates
`K` filled boxed plates
`M` unfilled boxes
`N` one rectangular plate
`P` alternate polarized
`+` adds a polarity sign
`+L` polarity sign to the left of drawing direction
arg3: `R`=reversed polarity
arg4: height (defaults `F: dimen_`/3, `C,P: dimen_`/4, `E,K: dimen_`/5)
arg5: wid (defaults `F:` *height**0.3, `C,P:` *height**0.4, `CP:` *height**0.8, `E,K:` *height*) (Section 4.2)

`case(`*i, alt1, alt2, ...*`)` dpictools Case statement for dpic; execute alternative *i*. Example: `case( 2, x=5, x=10, x=15 )` sets `x` to 10. Note: this is a macro so `$n` refers to the *n*-th argument of `case`.

`cbreaker(`*linespec*`, L|R, D|Th|TS, body name)`
cct circuit breaker to left or right, `D`: with dots; `Th`: thermal; `TS`: squared thermal; default body bounding box name is *Br* (Section 4.2)

`ccoax(at` *location*`, M|F,` *diameter, attributes*`)`
cct coax connector, `M`: male, `F`: female (Section 6)

`cct_minus` cct Negative sign for elements. Evaluates to `\scriptsize$-$` if the postprocessor is LaTeX or compatible, otherwise to `-`. A font, *libertinus* for example, may displace this character; a workaround is to redefine this macro to something like `\lower1.1ex\hbox{\scriptsize$-$}` to reposition it.

`cct_plus` cct Positive sign for elements. Evaluates to `\scriptsize$+$` if the postprocessor is LaTeX or compatible, to `svg_small(+)` if svg, otherwise to `+`. Redefine this macro to change the size or position of the symbol if required by the font in use.

`cct_init` cct initialize circuit-diagram environment (reads `libcct.m4`)

`centerline_(`*linespec, thickness|color, minimum long dash len, short dash len, gap len*
gen Technical drawing centerline

`c_fet(`*linespec*`,R,P)` cct left or right, plain or negated pin simplified MOSFET

`Cintersect(`*Pos1, Pos2, rad1, rad2,* `[R])`
gen Upper (lower if arg5=`R`) intersection of circles at *Pos1* and *Pos2*, radius *rad1* and *rad2*

`clabel(` *label*`,` *label*`,` *label*`,` *relative position*`,` *block name*`)`
cct Triple label along the drawing axis of the body of an element in the current direction (Section 5.1). Labels are placed at the beginning, centre, and end of the last `[]` block (or a `[]` block named or enumerated in arg5). Each label is treated as math by default, but is copied literally if it is in double quotes or sprintf. *Arg4* can be `above, below, left,` or `right` to supplement the default relative position.

`cm__` gen absolute centimetres

`cmyktorgb(`*c, m, y, k, r, g, b*`)` dpictools cmyk values in percent, i.e., 0 to 100, to rgb.

`consource(`*linespec*`, V|I|tv|v|ti|i|P, R,` *attributes*`)`

cct — controlled source or sensor with alternate forms; `V`: voltage; `I`: current; `v`: voltage type 2; `tv`: voltage type 3; `i`: current type 2; `ti`: current type 3; `P`: proximity sensor; `R`: reversed polarity. Body internal locations N, S, E, W, and C are defined. Arg 4 can be used to modify the body or to add internal symbols, e.g.
`consource(,,,fill_(0.9); "S" at C)` (Section 4.2)

`ColoredV(box|circle|ellipse,(`*r,g,b*`)|((`*colorseq*`))[:`*nlines*`],`*attributes*`)`

gen — box (default), circle, or ellipse in a `[]` block. If arg2 is blank then all formatting is in arg3; if parenthesized r,g,b, the object is shaded top to bottom white to the specified rgb color; if a double-parenthesized *colorseq* then the *colorseq* defines the internal shading top to bottom. A *colorseq* is of the form
0,r0,g0,b0,
frac1,r1,g1,b1,
frac2,r2,g2,b2,
...
1,rn,gn,bn
with $0 < frac1 < frac2 \ldots 1$. The number of *colorseq* lines can be specified with the colon (default `height/(line thickness)*2`). Examples:
```
ColoredV(circle,(1,0,0));
ColoredV(ellipse,(1,0.04,1),wid 0.75 ht 1 \
  outlined "magenta" "Goodbye");
ColoredV(box,((0,1,1,0, 1,0,0,1)):50,
  outlined "blue" rad 0.1).
```

`contact(`*chars*`)` — cct — single-pole contact: `O`: normally open
`C`: normally closed (default)
`I`: open circle contacts
`P`: three position
`R`: right orientation
`T`: T contacts
`U`: U contacts (Section 6)

`contacts(`*count, chars*`)` — cct — multiple ganged single-pole contacts: `P`: three position
`O`: normally open
`C`: normally closed
`D`: dashed ganging line over contact armatures `I`: open circle contacts
`R`: right orientation
`T`: T contacts
`U`: U contact lines parallel to drawing direction (Section 6)

`contline(`*line*`)` — gen — evaluates to `continue` if processor is **dpic**, otherwise to first arg (default `line`)

`copy3(`*vector1,vector2*`)` — dpictools — Copy vector1 into vector named by arg2.

`copythru(`*dpic macro name,* `"`*file name*`")` — dpictools — Implements the gpic `copy` *filename* `thru` *macro-name* for file data separated by commas, spaces, or tabs.

`corner(`*line thickness,attributes,turn radians*`)`
gen — Mitre (default filled square) drawn at end of last line or at a given position. arg1 default: current line thickness; arg2: e.g. `outlined` *string*; if arg2 starts with `at` *position* then a manhattan (right-left-up-down) corner is drawn; arg3= radians (turn angle, +ve is ccw, default $\pi/2$). The corner is enclosed in braces in order to leave `Here` unchanged unless arg2 begins with `at` (Section 7)

`Cos(`*integer*`)` — gen — cosine function, *integer* degrees

`cosd(`*arg*`)` — gen — cosine of an expression in degrees

`Cosine(` *amplitude*`,` *freq*`,` *time*`,` *phase* `)`
gen — function $a \times \cos(\omega t + \phi)$

`cross3(`*vec1, vec2, vec3*`)` — dpictools — The 3-vector cross product *vec3* = *vec1* × *vec2*.

`cross3D(`*x1,y1,z1,x2,y2,z2*`)` — 3D — cross product of two triples

`cross(at` *location, size*`|`*keys*`)` — gen — Plots a small cross. The possible key-value pairs are: `size=`*expr*`;`, `line=`*attributes*`;`

`crossover(`*linespec*`,` `[L|R][:`*line attributes*`],` *Linename1, Linename2, ...*`)`
cct — line jumping left or right over ordered named lines (Section 6.1)

`crosswd_` — gen — cross dimension

`csdim_` — cct — controlled-source width

D

`dabove(at` *location*`)` — darrow — above (displaced dlinewid/2)

`dac(`*width,height,nIn,nN,nOut,nS*`)`
cct — DAC with defined width, height, and number of inputs `In`*i*, top terminals `N`*i*, ouputs `Out`*i*, and bottom terminals `S`*i* (Section 9)

`Darc(`*center position*`,` *radius*`,` *start radians*`,` *end radians*`,` *parameters*`)`
darrow — Wrapper for `darc`. CCW arc in `dline` style, with closed ends or (dpic only) arrowheads. Semicolon-separated *parameters*:
`thick=`*value*;
`wid=`*value*;
`ends=` *x*`-`, `-`*x*, *x*`-`*x*, `->`, *x*`->`, `<-`, `<-`*x*, `<->`, where *x* is `|` or (half-thickness line) `!`.

`darc(`*center position*`,` *radius*`,` *start radians*`,` *end radians*`,` *dline thickness*`,` *arrowhead wid*`,` *arrowhead ht*`,` *end symbols*`,` *outline attributes*`,` *inner attributes*`)`
darrow See also `Darc`. CCW arc in `dline` style, with closed ends or (dpic only) arrowheads. Permissible *end symbols*: *x*`-`, `-`*x*, *x*`-`*x*, `->`, *x*`->`, `<-`, `<-`*x*, `<->` where *x* is `|` or (half-thickness line) `!`. An inner arc is drawn overlaying the outer arc. Example: `darc(,,,,,,,,outlined "red",outlined "yellow")`.

`Darlington(L|R,`*chars*`)` cct Composite Darlington pair Q1 and Q2 with internal locations E, B, C; Characters in *arg2:*
E= envelope
P= P-type
B1= internal base lead
D= damper diode
R1= Q1 bias resistor; E1= ebox
R2= Q2 bias resistor; E2= ebox (require R1 or E1)
Z= zener bias diode (Section 6.1)

`darrow(`*linespec*`, t, t,` *width*`,` *arrowhd wd*`,` *arrowhd ht*`,` *parameters*`,` *color attributes*`)`
darrow See also `Darrow`. Double arrow, truncated at beginning (arg2=`t`) or end (arg3=`t`), specified sizes, with arrowhead and optional closed stem. The parameters (arg7) are *x*`-` or `->` or *x*`->` or `<-` or `<-`*x* or `<->` where *x* is `|` or `!`. The `!-` or `-!` parameters close the stem with half-thickness lines to simplify butting to other objects. The color attributes are, e.g., `outlined "`*color*`" shaded "`*color*`"`. Example: `linethick=5; darrow(down_ 2,,,0.5,0.75,0.75,|,outlined "red")`.

`Darrow(`*linespec*`,` *parameters*`)` darrow Wrapper for `darrow`. Semicolon-separated *parameters*:
`S;`, `E;` truncate at start or end by dline thickness/2
`thick=`*val*; (total thicknes, ie width)
`wid=`*val*; (arrowhead width)
`ht=`*val*; (arrowhead height)
`ends=` *x*`-`*x* or `-`*x* or *x*`-` where *x* is `!` (half-width line) or `|` (full-width line).
Examples: `define('dfillcolor','1,0.85,0')` `linethick=5; rgbdraw(1,0,0,Darrow(down_ 2,thick=0.5; wid=0.75; ht=0.75; ends=|->))`, which is equivalent to `Darrow(down_ 2,thick=0.5; wid=0.75; ht=0.75; ends=|->; outline="red")`.

`darrow_init` darrow Initialize darrow drawing parameters (reads library file `darrow.m4`)

`dashline(`*linespec*`,`*thickness*`|`*color*`|<->|->|<-,` *dash len, gap len*`,G)`
gen Dashed line with dash at end (`G` ends with gap). Dashes are adjusted to fit with given gap length. Dpic only.

`dbelow(at` *location*`)` darrow below (displaced dlinewid/2)

`dcosine3D(`*i,x,y,z*`)` 3D extract i-th entry of triple x,y,z

`DCsymbol(at` *position, len, ht,*`U|D|L|R|`*degrees*`)`
cct A DC symbol (a dashed line below a solid line). The current drawing direction is default, otherwise Up, Down, Left, Right, or at *degrees* slant; e.g., `source(up_dimen_); { DCsymbol(at last [],,,R) }` (Section 4.2)

`DefineCMYKColor(`*color-name, c, m, y, k*`)`
dpictools Like `DefineRGBColor` but takes arguments in percent, i.e., the range $[0, 100]$. Define dpic macro *colorname* according to the postprocessor specified by dpic command-line option. The macro evaluates to a string.

`DefineHSVColor(`*color-name, h, s, v*`)`
dpictools Like `DefineRGBColor` but takes argument h in the range $[0, 360]$, s in $[0, 1]$, and v in $[0, 1]$. Define dpic macro *colorname* according to the postprocessor specified by dpic command-line option. The macro evaluates to a string.

`DefineRGBColor(`*color-name, r, g, b*`)`
dpictools Arguments are in the range 0 to 1. Define dpic macro *colorname* according to the postprocessor specified by dpic command-line option. The macro evaluates to a string.

`definergbcolor(`*color-name, r, g, b*`)`
gen Arguments are in the range 0 to 1. Define color name according to the postprocessor. Similar to dpictools `DefineRGBColor` but the color name is an m4 macro, not a string.

`delay(`*linespec*`,`*size*`,`*attributes*`)` cct delay element (Section 4.2)

`delay_rad_` cct delay radius

`deleminit_` darrow sets drawing direction for dlines

`Deltasymbol(at` *position, keys,* `U|D|L|R|`*degrees, attributes*`) (default U for up)`
cct Delta symbol for power-system and other diagrams. *keys:* `size=`*expression;* `type=C[losed]|E[xtended]|O[pen]`; `E` or `O` draws an "extended" or "open" symbol).
Arg3 sets drawing direction and Arg4 contains attributes of the drawn line object

`Demux(`*n*`,`*label*`, [L][B|H|X][N[`*n*`]|S[`*n*`]][[N]OE],` *wid*`,`*ht*`,`*attributes*`)`
log binary demultiplexer, n inputs
Arg5 is of the form *expr*`[:`*expr*`]`, i.e. right (output)-side height optionally followed by left (input)-side height;
`L` reverses input pin numbers
`B` displays binary pin numbers
`H` displays hexadecimal pin numbers
`X` do not print pin numbers
`N[`*n*`]` puts Sel or Sel0 .. Sel*n* at the top (i.e., to the left of the drawing direction)
`S[`*n*`]` puts the Sel inputs at the bottom (default) `OE` (`N`=negated) `OE` pin (Section 9)

`dend(at` *location*`,` *line thickness*`|`*attributes*`)`
darrow Close (or start) double line (Note specifying `dends=` for `Dline` is a similar function. Arg2 is dline thickness or atributes:
`thick=`*expression*`;` (dline thickness in drawing units)
`outline=(`*r*,*g*,*b*`)|"`*color*`";`

`d_fet(`*linespec*`,R,P,E|S)` cct left or right, N or P depletion MOSFET, envelope or simplified (Section 6.1)

`dfillcolor` darrow dline fill color (default white)

`diff3(`*vec1, vec2, vec3*`)` dpictools The 3-vector subtraction *vec3* = *vec1* − *vec2*.

`dfitcurve(`*Name, n, linetype, m*`)`
dpictools Draw a spline through *Name*`[m], ...`*Name*`[n]` with attribute *linetype* `dotted,` for example. The calculated control points *P[i]* satisfy approximately: *P[0] = V[0]; P[i-1]/8 + P[i]*3/4 + P[i+1]/8 = V[i]; P[n] = V[n].* See m4 macro `fitcurve.`

`dfitpoints(`*V,n,m,P,mp*`)` dpictools Compute the control locations `P[mP], P[mP+1]...` for the spline passing throught points `V[m]...V[n].` Used by macro `dfitcurve`.

`diff3D(`*x1,y1,z1,x2,y2,z2*`)` 3D difference of two triples

`diff_(`*a*,*b*`)` gen difference function

`dimen_` cct size parameter for scaling circuit element bodies (Section 12.1)

`dimension_(`*linespec*`,` *offset*`,` *label*`,` `D|H|W|`*blank width*`,` *tic offset*`,` `<- | ->)`
gen macro for dimensioning technical drawings; Arg1 defines the attributes of an invisible line: `line invis` arg1.
Arg2 is the sideways displacement (possibly negative) of the dimension arrows from the line.
Arg3: label, normally a number or number with unit symbol but if arg3 begins with `[` then it is copied verbatim.
Arg4: if arg3 is `s_box(...)` or `rs_box(...)` and arg4 is one of `D,H,W` then arg4 means:
`D:` blank width is the diagonal length of arg3;
`H:` blank width is the height of arg3 + `textoffset*2`;
`W:` blank width is the width of arg3 + `textoffset*2`;
otherwise arg4 is the absolute blank width.
Arg5 is `->` `|` `<-` to designate a single arrowhead at the end or start of the reference line; otherwise both arrowheads are drawn by default.

`diode(`*linespec*`, B|b|CR|D|G|L|LE[R]|P[R]|S|Sh|SI|T|U|V|v|w|Z|z|`*chars*`, [R][E])`

cct — diode: `B`: bi-directional
`b`: bi-directional with outlined zener crossbar
`CR`: current regulator
`D`: diac
`G`: Gunn
`L`: open form with centre line
`LE[R]`: LED [right]
`P[R]`: photodiode [right]
`S`: Schottky
`Sh`: Shockley
`SI`: SIDAC (see sidac)
`T`: tunnel
`U`: limiting
`V`: varicap
`v`: varicap (curved plate)
`w`: varicap (reversed polarity)
`Z`: zener
`z`: zener with angled centre bar
appending `K` to arg 2 draws open arrowheads; arg 3: `R`: reversed polarity, `E`: enclosure (Section 4.2)

`dir_` — darrow — used for temporary storage of direction by darrow macros

`distance(`*Position 1*`, `*Position2*`)`

gen — distance between named positions

`distance(`*position*`, `*position*`)` — gen — distance between positions

`dlabel(`*long*`,`*lat*`,`*label*`,`*label*`,`*label*`,`*chars*`)`

cct — general triple label; *chars: X* displacement *long*, *lat* with respect to the drawing direction is from the centre of the last line rather than the centre of the last `[ ]`; L,R,A,B align labels ljust, rjust, above, or below (absolute) respectively (Section 5.1)

`dleft(at `*position*`, `*line thickness*`, `*attributes*`)`

darrow — Double line left turn 90 degrees. Attributes can be `outline=(`*r, g, b*`)|"`*color*`"; innershade=(`*r, g, b*`)|"`*color*`"`; where rgb values in parentheses or a defined color is specified.

`Dline(`*linespec*`, `*parameters*`)` — darrow — Wrapper for `dline`. The semicolon-separated *parameters* are:
`S;`, `E;` truncate at start or end by dline thickness/2;
`thick=`*val*`;` (total thicknes, ie width);
`outline=`*color*`;` (e.g., `"red"` or `(1,0,0)`),
`innershade=`*color*`;` (e.g., `(0,1,1)` or `"cyan"`),
`name=`*Name*`;`,
`ends=`*x*`-`*x* or `-`*x* or *x*`-` where *x* is `!` (half-width line) or `|`; (full-width line).

`dline(`*linespec*`,t,t,`*width*`,`*parameters*`)`
darrow See also `Dline`. Double line, truncated by half width at either end, closed at either or both ends. *parameters*= *x*`-`*x* or `-`*x* or *x*`-` where *x* is `!` (half-width line) or `|` (full-width line).

`dlinewid` darrow width of double lines

`dljust(at` *location*`)` darrow ljust (displaced dlinewid/2)

`dna_` cct internal character sequence that specifies which subcomponents are drawn

`dn_` gen down with respect to current direction

`dot3(`*vec1, vec2*`)` dpictools Expands to the dot (scalar) product of the two 3-vector arguments: ($1[1] · $2[1] + $1[2] · $2[2] + $1[3] · $2[3]).

`dot3D(`*x1,y1,z1,x2,y2,z2*`)` 3D dot product of two triples

`dot(at` *location*`,`*radius*`|`*keys*`,`*fill*`)`
gen Filled circle (third arg= gray value: 0=black, 1=white). The possible key-value pairs are: `rad=`*expr*`;` and `circle=`*attributes*`;`

`dotrad_` gen dot radius

`down_` gen sets current direction to down (Section 5)

`dpquicksort(`*array name, lo, hi, ix*`)`
dpictools Given array *a[lo:hi]* and index array *ix[lo:hi]* = *lo, lo+1, lo+2,...hi*, sort *a[lo:hi]* and do identical exchanges on *ix*.

`dprot(`*radians, x, y*`)` dpictools Evaluates to a rotated pair (see m4 `rot_`).

`dprtext(`*degrees, text*`)` dpictools Rotated PStricks or pgf text in a `[]` box.

`dright(at` *position*`,` *line thickness*`,` *attributes*`)`
darrow Double line right turn 90 degrees. Attributes can be
`outline=(`*r, g, b*`)|"`*color*`";`
`innershade=(`*r, g, b*`)|"`*color*`";`
where rgb values in parentheses or a defined color is specified.

`drjust(at` *location*`)` darrow rjust (displaced dlinewid/2)

`dswitch(`*linespec*`, L|R, W[ud]B` *chars*`,` *attributes*`)`

cct Comprehensive IEEE-IEC single-pole switch: arg2=`R`: orient to the right of drawing dir
arg4 is a key-value sequence for the body of `GC` and `GX` options: `GC` keys: `diam, circle;GX` keys: `lgth, wdth, box, text`.
arg 3: blank means `WB` by default
`B`: contact blade open
`Bc`: contact blade closed
`Bm`: mirror blade
`Bo`: contact blade more widely open
`dB`: contact blade to the right of direction
`Cb`: circuit breaker function (IEC S00219)
`Co`: contactor function (IEC S00218)
`C`: external operating mechanism
`D`: circle at contact and hinge (`dD` = hinge only, `uD`: contact only)
`DI`: Disconnector, isolator (IEC S00288)
`E`: emergency button
`EL`: early close (or late open)
`LE`: late close (or early open)
`F`: fused
`GC`: disk control mechanism, attribs: `diam=`*expr*`;` `circle=`*circle attribs*`;`
`GX`: box control mechanism, attribs: `lgth=`*expr*`;` `wdth=`*expr*`;` `box=`*box attr*`;` `text=`*char*`;`
`H`: time delay closing
`uH`: time delay opening
`HH`: time delay opening and closing
`K`: vertical closing contact line use `WdBK` for a normally-closed switch
`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
`Pr[T|M]`: proximity (touch-sensitive or magnetically controlled)
`R`: time-delay operating arm
`Sd`: Switch-disconnector
`Th`: thermal control linkage
`Tr`: tripping
`W`: baseline with gap
`Y`: pull switch
`Z`: turn switch (Section 4.2)

`dtee([L|R],` *line thickness*`,` *attributes*`)`
darrow Double arrow tee junction with tail to left, right, or (default) back along current direction, leaving the current location at the tee centre; e.g., `dline(right_,,t); dtee(R); { darrow(down_,t) }; darrow(right_,t)`. The attributes are `thick=`*expr*`;` (line thickness in drawing units), `innershade=`(*r,g,b*)`|"`*color*`";` `outline=`(*r,g,b*)`|"`*color*`";`.

`dtor_` gen degrees to radians conversion constant

`dturn(`*degrees ccw*`,`*line thickness*`,` *attributes*`)`
darrow Tturn dline arg1 degrees left (ccw). Attributes can be `outline=`(*r, g, b*)`|"`*color*`"; innershade=`(*r, g, b*)`|"`*color*`";` where rgb values in parentheses or a defined color is specified.

E

`earphone( U|D|L|R|`*degrees, size*`)`
cct earphone, *In1* to *In3* defined (Section 6)

`ebox(`*linespec*`,`*lgth*`,`*wdth*`,`*fill value*`,` *box attributes*`)`
cct two-terminal box element with adjustable dimensions and fill value 0 (black) to 1 (white). *lgth* (length) and *wdth* (width) are relative to the direction of *linespec*. Alternatively, argument 1 is the *linespec* and argument 2 is a semicolon-separated sequence of key=value terms. The possible keys are `lgth, wdth, text, box`, e.g., `lgth=0.2; text="XX"; box=shaded "green"` (Section 4.2)

`E__` gen the constant *e*

`e_` gen .e relative to current direction

`e_fet(`*linespec*`,R,P,E|S)` cct left or right, N or P enhancement MOSFET, normal or simplified, without or with envelope (Section 6.1)

`elchop(`*Name1*`,`*Name2*`)` gen `chop` for ellipses: evaluates to `chop` *r* where *r* is the distance from the centre of ellipse Name1 to the intersection of the ellipse with a line to location Name2; e.g., `line from A to E elchop(E,A)`

`eleminit_(`*linespec*`)` cct internal line initialization

`elen_` cct default element length

`ellipsearc(`*width, height, startangle, endangle, rotangle,* `cw|ccw`*, line attributes*`)`
gen Arc of a rotated ellipse in a `[ ]` block. Angles are in radians. Arg5 is the angle of the width axis; e.g., `ellipsearc(2,1,0,pi_,pi_/4,,dashed ->)`. Internal locations are `Start, End, C` (for centre).

`em_arrows(`*type*`|`*keys, angle, length*`)`
cct — Radiation arrows: *type* `N|I|E [D|T]`:
`N`: nonionizing,
`I`: ionizing,
`E`: simple;
`D`: dot on arrow stem;
`T`: anchor tail;
keys: `type=`*chars* as above;
`angle=`*degrees*; (absolute direction)
`lgth=`*expr*;
`sep=`*expr*; arrow separation (Section 4.2)

`endshade` — gen — end gray shading, see `beginshade`

`Equidist3(`*Pos1, Pos2, Pos3, Result, distance*`)`
gen — Calculates location named *Result* equidistant from the first three positions, i.e. the centre of the circle passing through the three positions. If arg5 is nonblank, it is returned equated to the radius.

`expe` — gen — exponential, base e

F

`f_box(`*boxspecs*`,`*text*`,`*expr1*`,`$\cdots$`)` — gen — like `s_box` but the text is overlaid on a box of identical size. If there is only one argument then the default box is invisible and filed white (Section 14)

`Fector(`*x1,y1,z1,x2,y2,z2*`)` — 3D — vector projected on current view plane with top face of 3-dimensonal arrowhead normal to x2,y2,z2

`Fe_fet(`*linespec*`,R,`*chars*`)` — cct — FET with superimposed ferroelectric symbol. Args 1 to 3 are as for the `mosfet` macro (Section 6.1)

`FF_ht` — cct — flipflop height parameter in `L_units`, default 18

`FF_wid` — cct — flipflop width parameter in `L_units`, default 12

`fill_(`*number*`)` — gen — fill macro, 0=black, 1=white (Section 6.1)

`findroot(`*function name, left bound, right bound, tolerance, variable*`))`
dpictools — Solve *function*(x) = 0 by the method of bisection. The calculated value is assigned to the variable named in the last argument (Section 2.2). Example: `define parabola { $2 = ($1)^2 - 1 }; findroot( parabola, 0, 2, 1e-8, x )`.

`fitcurve(V, n,` *attributes*`, m (default 0))`
gen — Draw a spline through positions V[m], ... V[n]: Works only with dpic.

`FlipFlop(D|T|RS|JK,`*label*`,`*boxspec*`,`*pinlength*`)`
log — flip-flops, *boxspec* e.g., ht x wid y (Section 9)

`FlipFlopX`(*boxspec, label, leftpins, toppins, rightpins, bottompins, pinlength*)
log General flipflop. Arg 1 modifies the box (labelled Chip) default specification. Each of args 3 to 6 is null or a string of *pinspecs* separated by semicolons (`;`). A *Pinspec* is either empty or of the form `[`*pinopts*`]:[`*label*`[:`*Picname*`]]`. The first colon draws the pin. Pins are placed top to bottom or left to right along the box edges with null *pinspecs* counted for placement. Pins are named by side and number by default; eg `W1, W2, ..., N1, N2, ..., E1, ..., S1, ...` ; however, if `:`*Picname* is present in a *pinspec* then *Picname* replaces the default name. A *pinspec* label is text placed at the pin base. Semicolons are not allowed in labels; use, e.g., `\char59{}` instead. To put a bar over a label, use `lg_bartxt(`*label*`)`. The *pinopts* are `[N|L|M][E]`; `N`: pin with not circle; `L`: active low out; `M`: active low in; `E`: edge trigger (Section 9).
Optional arg 7 is the length of pins

`foreach_(‘`*variable*`’,`*actions*`,`*value1, value2, ...*`)`
gen Clone of Loopover_ by a different name: Repeat *actions* with *variable* set successively to *value1, value2, ...*, setting macro `m4Lx` to 1, 2, ..., terminating if *variable* is nul

`for_(`*start*`,`*end*`,`*increment*`,‘`*actions*`’)`
gen integer for loop with index variable `m4x` (Section 8)

`FTcap(`*chars*`)` cct Feed-through capacitor; example of a composite element derived from a two-terminal element. Defined points: *.Start, .End, .C, .T1, .T2, T*
Arg 1: `A`: type A (default), `B`: type B, `C`: type C (Section 6)

`fuse(`*linespec, type, wid, ht, attributes*`)`
cct fuse symbol, type= `A|B|C|D|S|HB|HC|SB` or `dA=D` (Section 4.2)

G

`gap(`*linespec*`,`*fill*`,A)` cct gap with (filled) dots, A=chopped arrow between dots (Section 4.2)

`gen_init` gen initialize environment for general diagrams (customizable, reads `libgen.m4`)

`g_fet(`*linespec*`,R,P,`*shade spec*`)`
cct left or right, N or P graphene FET, without or with shading (Section 6.1)

`g_` gen green color value

`G_hht` log gate half-height in `L_unit`s, default 3

`geiger(`*linespec*`, r, `*diameter*`, R, `*body attributes, body name*`)`
cct Wrapper that calls `source` with identical arguments except arg2, which is blank or `r` for right orientation.

`gpolyline_(`*fraction*`, `*location*`, ...)`
gen — internal to `gshade`

`graystring(`*gray value*`)` gen — evaluates to a string compatible with the postprocessor in use to go with `colored`, `shaded`, or `outlined` attributes. (PSTricks, metapost, pgf-tikz, pdf, postscript, svg). The argument is a fraction in the range $[0, 1]$; see `rgbstring`

`grid_(`*x*`,`*y*`)` log — absolute grid location

`ground(at` *location*`, T|`*stem length*`, N|F|S|L|P[A]|E, U|D|L|R|`*degrees*`)`
cct — ground, without stem for 2nd arg = T;
`N`: normal,
`F`: frame,
`S`: signal,
`L`: low-noise,
`P`: protective,
`PA`: protective alternate,
`E`: European; up, down, left, right, or angle from horizontal (default -90)
(Section 6)

`gshade(`*gray value*`,A,B,...,Z,A,B)`
gen — (Note last two arguments). Shade a polygon with named vertices, attempting to avoid sharp corners

`gyrator(`*box specs,space ratio,pin lgth,*`[N][V])`
cct — Gyrator two-port wrapper for `nport`, `N` omits pin dots; `V` gives a vertical orientation (Section 6)

H

`hatchbox(`*boxspec,hashsep,hatchspec,angle*`)` or `hatchbox(`*keys*`)`
gen — If Arg1 contains keys then a box is drawn in the current direction or as specified by `boxdir`; otherwise the box is drawn to the right. The hatch lines are at `angle` with respect to the current direction (default 45 degrees).
Defined keys are:
`wid=`*expr*`;`
`ht=`*expr*`;`
`box=`*attributes*`;` (e.g. `dashed outline "`*color*`"`)
`hatchsep=`*expr*`;`
`hatchspec=`*attributes*`;`
`angle=`*degrees*`;`
`boxdir=`*degrees*`;`
e.g., `hatchbox(outlined "blue",,dashed outlined "green" thick 0.4);`
also `define mycolor {rgbstring(1,0.2,0.5)};`
`hatchbox(box=dashed outlined mycolor)`

`Header(1|2,`*rows,wid,ht,box attributes*`)`
log — Header block with 1 or 2 columns and square Pin 1: arg1 = number of columns; arg2 = pins per column; arg3,4 = custom wid, ht; arg5 = e.g., `fill_(0.9)` (Section 6)

`HeaderPin(`*location, type, Picname*`,n|e|s|w,`*length*`)`
log General pin for `Header` macro; arg 4 specifies pin direction with respect to the current drawing direction)

`heatere(`*linespec, keys,* `[R][T])`
cct Heater element with curved sides (Section 4.2). `R` means right orientation; `T` truncates leads to the width of the body. The *keys* for the body are `lgth=`*expr*`; wdth=`*expr*`;` (default `lgth*2/5`); `cycles=`*expr*`;line=`*attributes*`;` (e.g., `dotted, dashed, outlined`)

`heater(`*linespec, ndivisions|keys, wid, ht, boxspec*`|[E[R][T]])`
cct Heater element (Section 4.2). If arg 5 contains `E,` draws an `heatere(`*linespec, keys,* `[R][T]),` otherwise a `heatert(`*linespec, nparts, wid, ht, boxspec*`)`

`heatert(`*linespec, nparts|keys, wid, ht, boxspec*`)`
cct Two-terminal rectangular heater element (Section 4.2). The *keys* for the body are `parts=`*expr*`; lgth=`*expr*`; wdth=`*expr*`;` (default `lgth*2/5`); `box=`*body attributes*`;` (e.g., `dotted, dashed, outlined, shaded`). Args 3–5 are unused if any key is given

`heatsink(at` *position, keys,* `U|D|L|R|`*degrees*`)`
cct Heatsink symbol drawn beside an element. *keys:* `lgth=`*expr; hght=expr; fin=attributes; base=attributes; fincount=expr;* Arg3: drawing direction (default `R`)

`hexadecimal_(`n`, [`m`])` gen hexadecimal representation of n, left padded to m digits if the second argument is nonblank

`hex_digit(`n`)` gen hexadecimal digit for $0 \le n < 16$

`H_ht` log hysteresis symbol dimension in `L_units`, default 2

`histbins(`*data-array name, n, min, max, nbins, bin array name*`)`
dpictools Generate the distribution of n values in *data-array*. If given, arg3 and arg4 specify maximum and minimum data values, otherwise they are calculated. Bins have index 0 to arg5-1.

`hlth` gen current line half thickness in drawing units

`hoprad_` cct hop radius in crossover macro

`hsvtorgb(`h, s, v, r, g, b`)` dpictools hsv color triple to rgb; h has range 0 to 360.

`ht_` gen height relative to current direction

I

`ifdpic(`*if true*`,`*if false*`)` gen test if dpic has been specified as pic processor

`ifgpic(`*if true*`,`*if false*`)` gen test if gpic has been specified as pic processor

`ifinstr`(*string*,*string*,*if true*,*if false*)
gen — test if the second argument is a substring of the first; also `ifinstr`(*string*,*string*,*if true*,*string*,*string*,*if true*, ... *if false*)

`ifmfpic`(*if true*,*if false*) gen — test if mfpic has been specified as pic post-processor

`ifmpost`(*if true*,*if false*) gen — test if MetaPost has been specified as pic post-processor

`ifpgf`(*if true*,*if false*) gen — test if Ti*k*z PGF has been specified as pic post-processor

`ifpostscript`(*if true*,*if false*) gen — test if Postscript (`dpic -r`) has been specified as pic output format

`ifpsfrag`(*if true*,*if false*) gen — Test if either `psfrag` or `psfrag_` has been defined. For postscript with psfrag strings, one or the other should be defined prior to or at the beginning of the diagram

`ifpstricks`(*if true*,*if false*) gen — test if PSTricks has been specified as post-processor

`ifroff`(*if true*,*if false*) gen — test if **troff** or **groff** has been specified as post-processor

`ifxfig`(*if true*,*if false*) gen — test if Fig 3.2 (`dpic -x`) has been specified as pic output format

`igbt`(*linespec*,`L|R,[L][[d]D]`) cct — left or right IGBT, L=alternate gate type, D=parallel diode, dD=dotted connections

`incircle`(*Vertex, Vertex, Vertex, InCentre, inrad*)
gen — Calculate the centre *InCentre* and radius *inrad* of a circle inscribed in the triangle with the thee vertices. e.g., `incircle(A,B,C,Ctr,rc); circle rad rc at Ctr`

`indicator`(*args 1–6 of* `source`, *keys*)
cct — Wrapper that calls `source` with up to 6 identical arguments and adds 4 rays. The *keys* in arg7 are: `len=`*expr*; `ray=`*attributes*; (of the 4 rays)

`inductor`(*linespec*, `W|L`, *cycles*, `M[n]|P[n]|K[n]`, *loop wid*)
cct — inductor, arg2: (default narrow), `W`: wide, `L`: looped;
arg3: number of arcs or cycles (default 4);
arg4: `M`: magnetic core, `P`: powder (dashed) core, `K`: long-dashed core, n=*integer* (default 2) number of core lines named *M4Core1, M4Core2,* ...;
arg5: loop width (default `L, W`: `dimen_`/5; other: `dimen_`/8) (Section 4.2)

`in__` gen — absolute inches

`inner_prod`(*linear obj*,*linear obj*)
gen — inner product of (x,y) dimensions of two linear objects

`integrator`(*linespec*,*size*) cct — integrating amplifier (Section 4.2)

`intersect_(`*line1*`.start,`*line1*`.end,` *line2*`.start,`*line2*`.end)`
gen intersection of two lines

`Intersect_(`*Name1*`,`*Name2*`)` gen intersection of two named lines

`Int_` gen corrected (old) gpic *int*() function

`IOdefs(`*linespec*`,`*label*`,[P|N]*,L|R)`
log Define locations *label*1, ... *label*n along the line; `P`: label only; `N`: with `NOT_circle`; `R`: circle to right of current direction

J

`jack(U|D|L|R|`*degrees*`,`*chars* `[;`*keys*`])`
cct arg1: drawing direction, normally R or L; character sequence arg2: `R`: right orientation, `X` external make or break contact points, one or more `L[M][B]` for L and auxiliary contacts with make or break points; `S[M][B]` for S and auxiliary contacts; `C[M][B]` for a centre contact (Section 6)

`j_fet(`*linespec*`,L|R,P,E)` cct left or right, N or P JFET, without or with envelope (Section 6.1)

`jumper(`*linespec, chars*`|`*keys*`)` cct Two-terminal solder jumper with named body parts. The *chars* character sequence specifies the jumper components, and normally begins with `C` and ends with `D`. The character `E` is an empty (blank) gap, `J` is a filled gap, `B` is a box component. The components are named *T1, T2, ...* Examples: `CED` is a simple open jumper (the default); `CJD` closed; `CEBED` three-contact open; `CJBED` three-contact open and closed. The *keys* are: `type=`*chars* as previously; `body=`*attributes* (e.g. `fill_(0.5)`); `wdth=`*expr*; `name=`*chars* (the body name) (Section 4.2)

K

`KelvinR(`*cycles*`,[R],`*cycle wid*`)` cct IEEE resistor in a `[ ]` block with Kelvin taps *T1* and *T2* (Section 6)

L

`l_coil(`*linespec, keys*`)` cct Ladder-diagram coil. The keys are: `size=`*expr*; (default `dimen_/3`), `type=P|N|S|NC`; (positive, negative, set latch, negated, default blank) (Section 4.2)

`l_contact(`*linespec, keys*`)` cct Ladder-diagram contact. The keys are: `wid=`*expr*; (default `dimen_/6`), `ht=`*expr*; (default `dimen_/3`), `type=P|N|NC`; (positive, negative, no contact, default blank) (Section 4.2)

`L_unit` log logic-element grid size

`lamp(`*linespec*`, [R][T])` cct Two-terminal incandescent lamp. `T` truncates leads to the body width. (Section 4.2)

`langle(`*Start, End*`)` gen Angle in radians from horizontal of the line from *Start* to *End*.

`larrow(`*label*`,->|<-,`*dist*`)` cct arrow *dist* to left of last-drawn 2-terminal element (Section 4.3)

`lbox(`*wid*`, `*ht*`, `*attributes*`, [r=`*expr*`])`
gen box oriented in current direction, arg 3= e.g. `dashed shaded "red"`. Arg 4 specifies box corner radius.

`LCintersect(`*line name, Centre, rad,* `[R], [`*Line start, End*`])`
gen First (second if arg4 is R) intersection of a line with a circle. Solves $|V.\texttt{start} + tV| = radius$ for t where V is the line. If arg1 is blank then the line start and end are given in arg5 and arg6.

`LCtangent(`*Pos1, Centre, rad,* `[R])`
gen Left (right if arg4=R) tangent point of line from Pos1 to circle at Centre with radius arg3

`left_` gen left with respect to current direction (Section 5)

`LEintersect(`*line name, Centre, ellipse wid, ellipse ht*`, [R], [`*Line start, End*`])`
gen First (second if arg5 is R) intersection of a line with an ellipse. If arg1 is blank then the line start and end are given in arg6 and arg7.

`length3(`*vector*`)` dpictools Euclidean length of 3-vector argument.

`length3D(x,y,z)` 3D Euclidean length of triple x,y,z

`LEtangent(`*Pos1, Centre, ellips wid, ellipse ht,* `[R])`
gen Left (right if arg5=R) tangent point of line from Pos1 to ellipse at Centre with given width and height

`lg_bartxt` log draws an overline over logic-pin text (except for xfig)

`lg_pin(`*location, label, Picname*`, n|e|s|w[L|M|I|O][N][E],` *pinno, optlen*`)`
log comprehensive logic pin;
label: text (indicating logical pin function, usually),
Picname: pic label for referring to the pin (line),
`n|e|s|w`: orientation (north, south, east, west),
`L`: active low out,
`M`: active low in,
`I`: inward arrow,
`O`: outward arrow,
`N`: negated,
`E`: edge trigger

`lg_pintxt` log reduced-size text for logic pins

`lg_plen` log logic pin length in in `L_units`, default 4

`LH_symbol([U|D|L|R|`*degrees*`][I],`*keys*`)`
log logic-gate hysteresis symbol; `I:` inverted. The keys are: `lgth=`*expr*`;`, `wdth=`*fraction*`;` i.e. body width =*fraction* × *height*

`lin_ang(`*line-reference*`[,d])` gen — the angle of a line or move from `.start` to `.end` of a linear object (in degrees if arg2=`d`)

`linethick_(`*number*`)` gen — set line thickness in points

`lin_leng(`*line-reference*`)` gen — length of a line, equivalent to *line-reference*`.len` with dpic

`ljust_` gen — ljust with respect to current direction

`llabel(` *label*, *label*, *label*, *relative position*, *block name*`)`
cct — Triple label on the left of the body of an element with respect to the current direction (Section 5.1). Labels are placed at the beginning, centre, and end of the last `[]` block (or a `[]` block named or enumerated in arg5). Each label is treated as math by default, but is copied literally if it is in double quotes or defined by sprintf. *Arg4* can be `above, below, left,` or `right` to supplement the default relative position.

`loc_(`*x*, *y*`)` gen — location adjusted for current direction

`log10E_` gen — constant $\log_{10}(e)$

`loge` gen — logarithm, base e

`log_init` log — initialize environment for logic diagrams (customizable, reads `liblog.m4`)

`loop(`*initial assignments, test, loop end, statements*`)`
dpictools — C-like loop. Commas in arg3 and arg4 must be in quotes or parentheses. Example:
`loop(i=1, i<=3, i+=1, print i)` prints 1, 2, 3.

`Loopover_('`*variable*`',`*actions*`,`*value1, value2, ...*`)`
gen — Repeat *actions* with *variable* set successively to *value1, value2, ...*, setting macro `m4Lx` to 1, 2, ..., terminating if *variable* is nul

`lpop(`*xcoord*, *ycoord*, *radius*, *fill*, *zero ht*`)`
gen — for lollipop graphs: filled circle with stem to (*xcoord,zeroht*)

`lp_xy` log — coordinates used by `lg_pin`

`lswitch(` *linespec*, `L|R`, *chars* `)`
cct — knife switch R=right orientation (default L=left); *chars*: [O|C][D][K][A] O=opening arrow; C=closing arrow; D=dots; K=closed switch; A=blade arrowhead (Section 4.2)

`lthick` gen — current line thickness in drawing units

`lt_` gen — left with respect to current direction

`LT_symbol(U|D|L|R|`*degrees*`,`*keys*`)`
log logic-gate triangle symbol. The keys are: `wdth=`*expr*`;`
M

`m4_arrow(`*linespec*`,`*ht*`,`*wid*`)` gen arrow with adjustable head, filled when possible

`m4dupstr(`*string*`,`*n*`,'`*name*`')` gen Defines *name* as *n* concatenated copies of *string*.

`m4lstring(`*arg1*`,`*arg2*`)` gen expand *arg1* if it begins with `sprintf` or `"`, otherwise *arg2*

`m4xpand(`*arg*`)` gen Evaluate the argument as a macro

`m4xtract('`*string1*`',`*string2*`)` gen delete *string2* from *string1*, return 1 if present

`manhattan` gen sets direction cosines for left, right, up, down

`Magn(`*length, height,* `U|D|L|R|`*degrees*`)`
cct magnetic action symbol.

`Max(`*arg, arg, ...*`)` gen Max of an arbitrary number of inputs

`memristor(`*linespec, wid, ht, attributes*`)`
cct memristor element (Section 4.2)

`microphone( A|U|D|L|R|`*degrees, size, attributes*`)`
cct microphone; if arg1 = A: upright mic, otherwise arg1 sets direction of standard microphone with *In1* to *In3* defined (Section 6)

`Min(`*arg, arg, ...*`)` gen Min of an arbitrary number of inputs

`Mitre_(`*Line1,Line2,length,line attributes*`)`
gen e.g., `Mitre_(L,M)` draws angle at intersection of lines L and M with legs of length arg3 (default `linethick bp__/2`); sets `Here` to intersection (Section 7)

`mitre_(`*Position1,Position2,Position3,length,line attributes*`)`
gen e.g., `mitre_(A,B,C)` draws angle ABC with legs of length arg4 (default `linethick bp__/2`); sets `Here` to Position2 (Section 7)

`mm__` gen absolute millimetres

`mosfet(`*linespec*`,L|R,`*chars*`,E)` cct MOSFET left or right, included components defined by characters, envelope. arg 3 chars:
`[u][d]B:` center bulk connection pin
`D:` D pin and lead
`E:` dashed substrate
`F:` solid-line substrate
`[u][d]G:` G pin to substrate at source
`[u][d]H:` G pin to substrate at center
`L:` G pin to channel (obsolete)
`[u][d]M:` G pin to channel, u: at drain end, d: at source end
`[u][d]M`*n*`:` multiple gates G0 to G*n*
`[d]Py:` parallel diode, d=reversed
`[d]Pz:` parallel zener diode, d=reversed
`O:` diode connection dots
`Q:` connect B pin to S pin
`R:` thick channel
`[u][d]S:` S pin and lead u: arrow up, d: arrow down
`[d]T:` G pin to center of channel d: not circle
`X:` XMOSFET terminal
`Z:` simplified complementary MOS (Section 6.1)

`Mux_ht` log Mux height parameter in `L_units`, default 18

`Mux(`*n*`,`*label*`, [L][B|H|X][N[`*n*`]|S[`*n*`]][[N]OE],` *wid*`,` *ht*`,` *attributes*`)`
log binary multiplexer, *n* inputs, Arg5 is of the form *expr*`[:`*expr*`]`, i.e. left (input)-side height optionally followed by right (output)-side height;
`L` reverses input pin numbers,
`B` display binary pin numbers,
`H` display hexadecimal pin numbers,
`X` do not print pin numbers,
`N[`*n*`]` puts Sel or Sel0 .. Sel*n* at the top (i.e., to the left of the drawing direction),
`S[`*n*`]` puts the Sel inputs at the bottom (default)
`OE` (`N`: negated) `OE` pin (Section 9)

`Mux_wid` log Mux width parameter in `L_units`, default 8

`Mx_pins` log max number of gate inputs without wings, default 6

N

`NAND_gate(`*n*`, [N][B], [`*wid*`, [`*ht*`]],` *attributes*`)`
log 'nand' gate, 2 or *n* inputs ($0 \le n \le 16$); `N`: negated inputs; `B`: box shape. Alternatively, `NAND_gate(`*chars*`, [B],` *wid*`,` *ht*`,` *attributes*`)`, where arg1 is a sequence of letters `P|N` to define normal or negated inputs. (Section 9)

`N_diam` log diameter of 'not' circles in `L_units`, default 1.5

`NeedDpicTools` gen executes `copy "HOMELIB_/dpictools.pic"` if the file has not been read

`neg_` gen unary negation

`ne_`	gen	.ne with respect to current direction
`n_`	gen	.n with respect to current direction
`norator(`*linespec*`,`*width*`,`*ht*`,`*attributes*`)`	cct	norator two-terminal element (Section 4.2)
`NOR_gate(`*n*`,N)`	log	'nor' gate, 2 or *n* inputs; `N`: negated input. Otherwise, arg1 can be a sequence of letters `P|N` to define normal or negated inputs. (Section 9)
`NOT_circle`	log	'not' circle
`NOT_gate(`*linespec*`,[B][N|n],`*wid*`,`*height*`,` *attributes*`)`	log	'not' gate. When *linespec* is blank then the element is composite and In1, Out, C, NE, and SE are defined; otherwise the element is drawn as a two-terminal element. arg2: `B`: box gate, `N`: not circle at input and output, `n`: not circle at input only (Section 9)
`NOT_rad`	log	'not' radius (`N_rad`) in absolute drawing units
`NPDT(`*npoles*`,[R])`	cct	Double-throw switch; *npoles:* number of poles; `R`: right orientation with respect to drawing direction (Section 6)
`nport(`*box spec*`;`*other commands, nw,nn,ne,ns,space ratio,pin lgth,style, other commands*`)`	cct	Default is a standard-box twoport. Args 2 to 5 are the number of ports to be drawn on w, n, e, s sides. The port pins are named by side, number, and by a or b pin, e.g., W1a, W1b, W2a, ... Arg 6 specifies the ratio of port width to interport space (default 2), and arg 7 is the pin length. Set arg 8 to N to omit the dots on the port pins. Arguments 1 and 9 allow customizations (Section 6)
`N_rad`	log	radius of 'not' circles in `L_units`, default `N_diam/2`
`nterm(`*box spec*`;`*other commands, nw,nn,ne,ns,pin lgth,style, other commands*`)`	cct	n-terminal box macro (default three pins). Args 2 to 5 are the number of pins to be drawn on W, N, E, S sides. The pins are named by side and number, e.g. W1, W2, N1, ... Arg 6 is the pin length. Set arg 7 to N to omit the dots on the pins. Arguments 1 and 8 allow customizations, e.g. `nterm(,,,,,,N,"$a$" at Box.w ljust,"$b$" at Box.e rjust, "$c$" at Box.s above)`
`nullator(`*linespec*`,`*width*`,`*ht*`,`*attributes*`)`	cct	nullator two-terminal element (Section 4.2)
`nw_`	gen	.nw with respect to current direction

`NXOR_gate(`*n*`,N)` log 'nxor' gate, 2 or *n* inputs; `N`: negated input. Otherwise, arg1 can be a sequence of letters `P|N` to define normal or negated inputs. The default output conforms to current ANSI standard by drawing short lines from the inputs to the gate main body. Original behaviour to omit them can be set by `define('XOR_off',-1)` either globally or for individual gates. (Section 9)

O

`opamp(`*linespec*`,`*label, label, size*`|`*keys, chars, other commands*`)` cct operational amplifier with −, + or other internal labels and specified size, drawn in a `[ ]` block. *chars:* `P` add power connections *V1* and *V2*, `R` swap *In1, In2 labels*, `T` truncated point. The internally defined positions are *W, N, E, S, C, Out, NE, SE, In, In2*, and the (obsolete) positions *E1* = *NE*, *E2* = *SE*. Instead of a size value, arg4 can be a key-value sequence. The keys are: `lgth=`*expr*`;`, `wdth=`*expr*`;`, `body=`*attributes*`;`, e.g., `body=shaded "`*color*`"`. (Section 6)

`open_arrow(`*linespec*`,`*ht*`,`*wid*`)` gen arrow with adjustable open head

`OR_gate(`*n*`,[N][B],` *wid, ht,attributes*`)` log Or gate, *n* inputs ($0 \le n \le 16$); arg2: `N`: negated inputs; `B`: box gate. Otherwise, arg1 can be a sequence of letters `P|N` to define normal or negated inputs. (Section 9)

`OR_gen(`*n*`,`*chars*`,[`*wid*`,[`*ht*`]],` *attributes*`)` log General OR gate: *n*=number of inputs ($0 \le n \le 16$);
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.
Otherwise, arg1 can be a sequence of letters `P|N` to define normal or negated inputs. If arg5 contains `shaded rgbstring(...)` the arguments of `rgbstring` may not contain parentheses.

`OR_rad` log radius of OR input face in `L_units`, default 7

P

`parallel_('`*elementspec*`','`*elementspec*`'...)` cct Parallel combination of two-terminal elements in a `[]` block. Each argument is a *quoted* elementspec of the form `[Sep=`*val*`;][`*Label*`:]`*element*`;[`*attributes*`]` where an *attribute* is of the form `[llabel(...);]|[rlabel(...);]|[b_current(...);]`. An argument may also be `series_(...)` or `parallel_(...)` *without* attributes or quotes. Sep=*val*; in the first branch sets the default separation of all branches to *val*; in a later element Sep=*val*; applies only to that branch. An element may have normal arguments but should not change the drawing direction. (Section 5.2)

`pconnex(R|L|U|D|`*degrees*`,`*chars, attributes*`)`
cct — power connectors, arg 1: drawing direction; *chars:*
`R` (right orientation)
`M|F` (male, female)
`A[B]|AC` (115V 3-prong, B: default box, C: circle)
`P` (PC connector)
`D` (2-pin connector)
`G|GC` (GB 3-pin)
`J` (110V 2-pin) (Section 6)

`pc__` — gen — absolute points

`perpto(`*Pos1, Line, Point*`)` — gen — *Point* is the label for the point on *Line* of the perpendicular from *Point* to *Line*.

`Pconn([−]`*n*`,U|D|L|R|`*degrees*`[:`*length*`], `*chars*`|`*keys*`)`
cct — Multiple `tconn` connectors nameed *T1* to *Tn* in a `[ ]` block. A negative arg1 reverses pin numbers *n* to 1 instead of 1 to *n*. Arg2 specifies the drawing direction (up, down, left, right, angle) and, if `:`*expression* is appended, the length of the pins. The permissible *chars* are as for `tconn`: `> | >> | < | << | A | AA | M | O | OF`. Type `O` draws a node (circle); `OF` a filled circle. Type `M` is a black bar; `A` is an open arc end; type `AA` a double open arc. Type `>` (the default) is an arrow-like output connector; `<` and `<<` input connectors. The keys are `type=`*chars* as above; `wdth=`*expr*; `lgth=`*expr*; `sep=`*expr*; `head=`*attributes except* `lgth, wdth.` The key `sep=` is the double-head separation. Additionally, the key `pitch=`*expr* specifies connector separation. (Section 6)

`PerpTo(`*Pos1, Pos2, Pos3*`)` — gen — The point between Pos2 and Pos3 of intersection of the perpendicular to Pos1, i.e., the perpendicular projection of Pos1 onto the line from Pos2 to Pos3.

`pi_` — gen — π

`plug(U|D|L|R|`*degrees*`,[2|3][R])`
cct — Phone plug; arg1: drawing direction; arg2: `R` right orientation, `2|3` number of conductors (Section 6)

`pmod(`*integer, integer*`)` — gen — +ve mod(M, N) e.g., `pmod`$(-3, 5) = 2$

`point_(`*angle*`)` — gen — (radians) set direction cosines

`Point_(`*integer*`)` — gen — sets direction cosines in degrees (Section 5)

`polar_(`*x*`,`*y*`)` — gen — rectangular-to polar conversion

`polygon(`*n*`,`*keys*`)` gen Regular polygon in a `[]` block. The keys are `line=`*line attributes*`;` (e.g., `dashed shaded "blue"`), `rot=`*degrees*`;` (angle of first internal vertex `V[0]`), `side|rad=`*expression*`;` size by side length or by radius. `radv=`*expression*`;` radius of rounded vertices. If this is nonzero then any fill has to be by `rgbfill(r,g,b,polygon(...))`. The internal defined points are the centre `C` and vertices `V[0]` ... `V[`*n*`]`.

`posarray(`*Name, Position1, Position2, ...*`)`
dpictools Populate a singly-subscripted array of positions: *Name*`[1]:`*Position1; Name*`[2]=`*Position2;* `....`

`posarray2(`*Name, expr, Position1, Position2, ...*`)`
dpictools Populate a doubly-subscripted array of positions: *Name*`[`*expr*`,1]=`*Position1; Name*`[`*expr*`,2]=`*Position2;* `....`

`potentiometer(`*linespec*`,`*cycles*`,`*fractional pos*`,`*length*`,`$\cdots$`)`
cct resistor with taps T1, T2, ... with specified fractional positions and lengths (possibly neg) (Section 6)

`print3D(x,y,z)` 3D write out triple for debugging

`prod_(`*a*`,`*b*`)` gen binary multiplication

`project(`*x*`,`*y*`,`*z*`)` 3D 3D to 2D projection onto the plane perpendicular to the view vector `View3D` with angles defined by `setview(`*azimuth, elevation, rotation*`)`.

`Proxim(`*size,* `U|D|L|R|`*degrees, attributes*`)`
cct proximity detector with fillable body.

`proximity(`*linespec*`)` cct proximity detector (= `consource(,P)`)

`psset_(`*PSTricks settings*`)` gen set PSTricks parameters

`PtoL(`*position*`,` `U|D|L|R|`*degrees*`,` *length*`)`
gen Evaluates to `from` *position* `to` *position* `+ Rect_(`*length, angle*`)` from the polar-coordinate data in the arguments

`pt__` gen TEX point-size factor, in scaled inches, (`*scale/72.27`)

`ptrans(`*linespec*`,` `[R|L])` cct pass transistor; `L=` left orientation (Section 6.1)

`pushkey_(`*string, key, default value,*`[N])`
gen Key-value definition. If *string* contains the substring *key*`=`*expr* then macro `m4`*key* is defined using `pushdef()` to expand to `(`*expr*`)`, or to `(`*default value*`)` if the substring is missing. Arg 1 can contain several such substrings separated by semicolons. If arg4 is nonblank, the parentheses are omitted. (Section 13.1)

`pushkeys_(`*string, key sequence*`)`

gen — Multiple key-value definitions. Arg2 is a semicolon-separated sequence of terms of the form *key*`:`*default-value*`[:N]` which must contain no semicolons and the default values contain no colons. A key may not be the tail of another key. Macro `pushkey_` is applied to each of the terms in order. Quote arg2 for robustness and, if an argument depends on a previous argument, add quotes to delay expansion; for example `pushkeys_(‘$1’,‘hght:0.5; wdth:m4‘’hght/2’)`. (Section 13.1)

`pvcell(`*linespec, width, height, attributes*`)`

cct — PV cell

`px__` — gen — absolute SVG screen pixels

R

`randn(`*array name, n, mean, stddev*`)`

dpictools — Assign *n* Gaussian random numbers in array *name*[1], *name*[2], ... *name*[*n*] with given mean and standard deviation.

`rarrow(`*label*`,->|<-,`*dist*`)` — cct — arrow *dist* to right of last-drawn 2-terminal element (Section 4.3)

`r_stub( at` *position, keys*`)` — cct — microstrip radial stub. The *keys* are:
`dir=U|D|L|R|`*degrees*`;` (default `U`) drawing direction (see `setdir_`)
`irad=`*expr*`;` inner radius
`orad=`*expr*`;` outer radius
`angle=`*expr*`;` (degrees) sector angle
`outline=`outline attributes; (but not thickness)
`fill=`internal shade color;
`stem=T[I]|`stem attributes; `T` means no stem, `TI` means no stem and no inner arc (Section 6)

`Rect_(`*radius*`,`*angle*`)` — gen — (deg) polar-to-rectangular conversion

`rect_(`*radius*`,`*angle*`)` — gen — (radians) polar-rectangular conversion

`reed(`*linespec, width, height, box attribues*`, [R][C])`

cct — Enclosed reed two-terminal contact; `R`: right orientation; `C`: closed contact; e.g., `reed(,,dimen_/5,shaded "lightgreen"` (Section 6)

`relaycoil`(*chars, wid, ht,* `R|L|U|D|`*degrees, attributes*)

	cct	chars: `X`: or default: external lines from A2 and B2; `AX`: external lines at positions A1,A3; `BX`: external lines at positions B1,B3; `NX`: no lines at positions A1,A2,A3,B1,B2,B3; `SO`: slow operating; `SOR`: slow operating and release; `SR`: slow release; `HS`: high speed; `S`: diagonal slash; `NAC`: unaffected by AC current; `AC`: AC current; `ML`: mechanically latched; `PO`: polarized; `RM`: remanent; `RH`: remanent; `TH`: thermal; `EL`: electronic (Section 6)

`relay`(*number of poles, chars, attributes*)

	cct	relay: n poles (default 1), *chars:* `O`: normally open, `C`: normally closed, `P`: three position, default double throw, `L`: drawn left (default), `R`: drawn right, `Th`: thermal. (Section 6)

`resetdir_`	gen	resets direction set by `setdir_`

`resetrgb`	gen	cancel `r_, g_, b_` color definitions

`resistor(`*linespec*`, cycles,` *chars*`,` *cycle wid*`)`
cct — resistor, number of cycles given by arg2 (default 3), *chars:*
`AC`: general complex element,
`E`: `ebox`,
`ES`: `ebox` with slash,
`EX`: `ebox` with full-size X,
`F`: FDNR (frequency-dependent negative resistor),
`Q`: offset,
`H`: squared,
`LD`: light-dependent,
`LDE`: light-dependent ebox,
`N`: IEEE (default),
`B`: not burnable,
`T`: thermistor,
`V`: varistor variant,
`R`: right-oriented with respect to drawing direction;
Arg4: *cycle width* (default `dimen_`/6.)
Alternative invocation:
`resistor(`*linespec*`,` *keys*`)`
The *keys* are: semicolon (;)-separated sequence of *key*=*value* pairs. Allowable keys are:
`type=`*chars*`;` as above,
`wdth=`*expression*`;` body width,
`cycles=`*integer expression*`;` (default 3),
`lgth=`*expression*`;` body length (default `(cycles)*(wdth)`),
`body=`*attributes*`;` for a box body (types `E, ES, AC`),
`env=`*attributes*`;` for the envelope (types `T, LD`).
(Section 4.2) (Section 13.1)

`resized(`*factor*`,'`*macro name*`',args)`
cct — scale the element body size by *factor*

`restorem4dir(['`*stack name*`'])`
gen — Restore m4 direction parameters from the named stack; default `'savm4dir_'`

`reversed('`*macro name*`',args)`
cct — reverse polarity of 2-terminal element

`rgbdraw(`*color triple*`,` *drawing commands*`)`
gen — color drawing for PSTricks, pgf, MetaPost, SVG postprocessors; (color entries are 0 to 1), see `setrgb` (Section 6.1). Exceptionally, the color of SVG arrows other than the default black has to be defined using the `outlined` *string* and `shaded` *string* constructs.

`rgbfill(`*color triple*`,` *closed path*`)`
gen — fill with arbitrary color (color entries are 0 to 1); see `setrgb` (Section 6.1)

`rgbstring`(*color triple or color name*)

gen — evaluates to a string compatible with the postprocessor in use to go with `colored`, `shaded`, or `outlined` attributes. (PSTricks, metapost, pgf-tikz, pdf, postscript, svg). The arguments are fractions in the range $[0, 1]$; For example, `box outlined rgbstring(0.1,0.2,0.7) shaded rgbstring(0.75,0.5,0.25)`. For those postprocessors that allow it, there can be one argument which is the name of a defined color. This macro can be fragile when used as an m4 macro argument. Then something like the following delays expansion:
```
define rgbpurp {rgbstring(0.5,0,1)};
curve(,,,rail=outlined rgbpurp)
```

`rgbtocmyk`(*r, g, b, c, m, y, k*) — dpictools — rgb to cmyk values in the range 0 to 100.

`rgbtohsv`(*r, g, b, h, s, v*) — dpictools — rgb color triple to hsv with *h* range 0 to 360.

`RightAngle`(*Pos1, Pos2, Pos3, line len, attributes*)

gen — Draw a right-angle symbol at *Pos2*, of size given by arg4. Arg5 = line attributes, e.g., `outlined "gray"` or e.g. to add a dot, `;dot(at last line.c)`

`right_` — gen — set current direction right (Section 5)

`rjust_` — gen — right justify with respect to current direction

`rlabel(` *label*`,` *label*`,` *label*`,` *relative position*`,` *block name*`)`

cct — Triple label on the right of the body of an element with respect to the current direction (Section 5.1). Labels are placed at the beginning, centre, and end of the last `[]` block (or a `[]` block named or enumerated in arg5). Each label is treated as math by default, but is copied literally if it is in double quotes or defined by sprintf. *Arg4* can be `above, below, left,` or `right` to supplement the default relative position.

`RotarySwitch(` *start degrees*`,` *end degrees*`,` *keys*`)`

cct — Rotary switch with poles drawn in a `ccw` arc from *start degrees* to *end degrees* and encased in a `[ ]` block. The keys are:
`poles=`*integer*`;` the number of peripheral poles
`circle=`*attributes*`;` such as rad, shaded, ...
`rad=`*expr*`;` arc radius of the poles, default `dimen_`
`wipers=`*wiperspec* `&` *wiperspec* `& ...;` A *wiperspec* is either `nil` or `B:`*degrees*`:`*length* and the latter two expressions can be omitted.
`wline=`*attributes*`;` wiper attributes (default `->`)
`segments=`*segspec* `&` *segspec* `& ...;` A *segspec* is *start deg*`:`*end deg* `:`*radius*`:`*thickness* and the latter two can be omitted.
The default is `RotarySwitch(-45:45,poles=4;wipers=B:45).` The centre dot is labeled *C*, the poles are *P1, P2,...*, the wipers *Wiper1, Wiper2,...*, and the segments *Seg1, Seg2,...*.

`rot3Dx(`*radians,x,y,z*`)` 3D rotates x,y,z about x axis

`rot3Dy(`*radians,x,y,z*`)` 3D rotates x,y,z about y axis

`rot3Dz(`*radians,x,y,z*`)` 3D rotates x,y,z about z axis

`rotbox(`*wid,ht,attributes*`,[r|t=`*val*`])`
gen box oriented in current direction in `[ ]` block; *attributes:* e.g. `dotted shaded "green".` Defined internal locations: N, E, S, W (and NE, SE, NW, SW if arg4 is blank). If arg4 is `r=`*val* then corners have radius *val*. If arg4 is `t=`*val* then a spline with tension *val* is used to draw a "superellipse," and the bounding box is then only approximate.

`rotellipse(`*wid,ht,attributes*`)` gen ellipse oriented in current direction in `[ ]` block; e.g. `Point_(45); rotellipse(,,dotted fill_(0.9)).` Defined internal locations: N, S, E, W.

`Rot_(`*position, degrees*`)` gen rotate position by degrees

`rot_(`*x, y, angle*`)` gen rotate *x,y* by theta radians

`round(at` *location,line thickness,attributes*`)`
gen filled circle for rounded corners; attributes=`colored "gray"` for example; leaves `Here` unchanged if arg1 is blank (Section 7)

`rpoint_(`*linespec*`)` gen set direction cosines

`rpos_(`*position*`)` gen Here + *position*

`r_` gen red color value

`rrot_(`*x, y, angle*`)` gen `Here + vrot_(`*x, y, cos(angle), sin(angle))*

`rs_box([angle=`*degrees*`;]` *text*`,`*expr1*`,`$\cdots$`)`
gen like `s_box` but the text is rotated by `text_ang`(default 90) degrees, unless the first argument begins with `angle=`*decimal number*`;`, in which case the number defines the rotation angle. Two or more args are passed to `sprintf()`. If the first argument begins with `angle=`*expr*`;` then the specified angle is used. The examples `define('text_ang',45); rs_box(Hello World)` and `rs_box(angle=45; Hello World)` are equivalent (Section 14), (Section 15)

`rsvec_(`*position*`)` gen Here + *position*

`r_text(`*degrees*`,`*text*`,at` *position*`)`
gen Rotate text by arg1 degrees (provides a single command for PSTricks, PGF, or SVG only) placed at position in arg3. The first argument is a decimal constant (not an expression) and the text is a simple string without quotes. (Section 14), (Section 15)

`rtod__`	gen	constant, degrees/radian
`rtod_`	gen	constant, degrees/radian
`rt_`	gen	right with respect to current direction
`rvec_(`*x*`,`*y*`)`	gen	location relative to current direction
`rvec_r(`*x*`,`*y*`)`	gen	Robust location relative to current direction for use in dpic loops

S

`sarrow(`*linespec*`,`*keys*`)`	gen	Single-segment, single-headed special arrows with *keys*: `type=O[pen]` (default) | `D[iamond]` | `C[rowfoot]` | `DI` (disk) | `P[lain]` | `PP[lain]` | `R[ight]` | `L[eft]` ; `wdth=`*expression*`;` (default `arrowwid`) `lgth=`*expression*`;` (default `arrowht`) `head=`*head attributes*`;` (e.g., `shaded`) `shaft=`*shaft attributes*`;` (default: head attributes) `hook=[L|R|LR]` (left, right, or double hook, default none) `name=`*Name*`;` (default *Sarrow_*) The `PP` key creates a doubled plain arrowhead (Section 13.1)
`savem4dir([`'*stack name*'`])`	gen	Stack m4 direction parameters in the named stack (default '`savm4dir_`')
`s_box(`*text*`,`*expr1*`,`$\cdots$`)`	gen	generate dimensioned text string using `\boxdims` from `boxdims.sty`. Two or more args are passed to `sprintf()` (default 90) degrees (Section 14)
`sbs(`*linespec, chars, label*`)`	cct	Wrapper to place an SBS thyristor as a two-terminal element with `[ ]` block label given by the third argument (Section 6.1)
`sc_draw(`*dna string, chars, iftrue, iffalse*`)`	cct	test if chars are in string, deleting chars from string
`scr(`*linespec, chars, label*`)`	cct	Wrapper to place an SCR thyristor as a two-terminal element with `[ ]` block label given by the third argument (Section 6.1)
`scs(`*linespec, chars, label*`)`	cct	Wrapper to place an SCS thyristor as a two-terminal element with `[ ]` block label given by the third argument (Section 6.1)
`s_dp(`*name*`,`*default*`)`	gen	depth of the most recent (or named) `s_box` (Section 14)

`series_(`*elementspec*`,` *elementspec*`, ...)`
cct — Series combination in a `[]` block of elements with shortened default length. Each argument is an elementspec of the form `[Sep=`*val*`;][`*Label*`:]`*element*`;[`*attributes*`]` where an *attribute* is of the form `[llabel(...);]|[rlabel(...);]|[b_current(...);]`. An argument may also be `series_(...)` or `parallel_(...)` *without* attributes or quotes. An element may have normal arguments but should not change the drawing direction. Internal points `Start`, `End`, and `C` are defined (Section 5.2)

`se_` — gen — .se with respect to current direction

`setdir_(R|L|U|D|`*degrees*`,` *default*`U|D|R|L|`*degrees*`)`
gen — store drawing direction and set it to up, down, left, right, or angle in degrees (reset by `resetdir_`). The directions may be spelled out, i.e., Right, Left, ... (Section 5.2)

`setkey_(`*string, key, default,*`[N])`
gen — Key-value definition, like `pushkey_()` but the resulting macro is defined using `define()` rather than `pushdef()`. (Section 13.1)

`setkeys_(`*string, key sequence*`)` — gen — Multiple key-value definition using `define()` rather than `pushdef()`. See macro `pushkeys_`. (Section 13.1)

`setrgb(`*red value, green value, blue value*`,[`*name*`])`
gen — define colour for lines and text, optionally named (default `lcspec`); (Section 6.1)

`setview(`*azimuth degrees*`,`*elevation degrees*`,` *rotation degrees*`)`
3D — Set projection viewpoint for the `project` macro. The view vector is obtained by looking in along the x axis, then rotating about $-x$, $-y$, and z in that order. The components `view3D1, view3D2,` and `view3D3` are defined, as well as positions `UPx_, UPy_,` and `UPz_` which are the projections of unit vectors `(1,0,0), (0,1,0),` and `(0,0,1)` respectively onto the plane.

`sfgabove` — cct — like above but with extra space

`sfgarc(`*linespec*`,`*text*`,`*text justification*`,cw|ccw,` *height scale factor*`,`*arc attributes*`)`
cct — Directed arc drawn between nodes, with text label and a height-adjustment parameter. Example: `sfgarc(from B to A,-B/M,below,,1.1,outlined "red")`

`sfgbelow` — cct — like below but with extra space

`sfg_init(`*default line len, node rad, arrowhd len, arrowhd wid*`), (reads libcct.m4)`
cct — initialization of signal flow graph macros

`sfgline(`*linespec*`,`*text*`,sfgabove|sfgbelow|ljust|rjust,`*line attributes*`)`
cct — Directed straight line chopped by node radius, with text label, e.g., `sfgline(,K/M,,dashed colored "orange")`

`sfgnode(at` *location*`,`*text*`,above|below,`*circle attributes*`)`

cct — small circle default white interior, with text label. The default label position is inside if the diameter is bigger than `textht` and `textwid`; otherwise it is `sfgabove.` Options such as color, fill, or line thickness can be given, e.g., `thick 0.8 outlined "red" shaded "orange"`.

`sfgself(at` *location*`, U|D|L|R|`*degrees*`,` *text label*`,` *text justification*`, cw|ccw,` *scale factor*`, [-> | <- | <->],` *attributes*`)`

cct — Self-loop drawn at an angle from a node, with text label, specified arrowheads, and a size-adjustment parameter. The attributes can set thickness and color, for example.

`shade(`*gray value*`,`*closed line specs*`)`

gen — Fill arbitrary closed curve. Note: when producing pdf via pdflatex, line thickness changes within this macro must be made via the `linethick` environment variable rather than by the `thickness` line attribute

`shadebox(box` *attributes, shade width*`)`

gen — Box with edge shading. Arg2 is in points. See also `shaded`

`shadedball(`*radius, highlight radius, highlight degrees, initial gray, final gray* `|` `(`*rf,gf,bf*`))`

3D — Shaded ball in `[ ]` box. The highlight is by default at *radius**3/5 and angle 110 deg (or arg2 deg); if setlight has been invoked then its azimuth and elevation arguments determine highlight position. Arg5 can be a parenthesized rgb color.

`ShadedPolygon(`*vertexseq, line attributes, degrees, colorseq*`)`

gen — Draws the polygon specified in arg1 and shades the interior according to arg4 by drawing lines perpendicular to the angle in arg3. The *vertexseq* is a colon (`:`) separated sequence of vertex positions (or names) of the polygon in cw or ccw order. A *colorseq* is of the form 0, r0,g0,b0, *frac1*,r1,g1,b1, *frac2*,r2,g2,b2, ... 1,rn,gn,bn with $0 < frac1 < frac2 \ldots 1$

`ShadeObject(`*drawroutine, n, colorseq*`)`

dpictools Fill an area in a `[]` block with graded color defined by *colorseq*, an indexed sequence of rgb colors: *frac0,r0,g0,b0, frac1,r1,g1,b1, ... fracn,rn,gn,bn* with $0 \le frac0 < frac1 < frac2 < \ldots fracn \le 1$. (Often $frac0 = 0$ and $fracn = 1$.) The dpic macro *drawroutine*(*frac, r, g, b*) typically draws a colored line and must be defined according to the area to be filled. This routine is called $n{+}1$ times for $frac = frac0,\ frac0 + 1/n \times (fracn - frac0)$, $frac0 + 2/n \times (fracn - frac0)$, $\ldots fracn$ (i.e., often $frac = 0, 1/n, 2/n, \ldots 1$) with rgb arguments interpolated in hsv space between *colorseq* points (which are specified in rgb-space). Example (shade a box with 101 graded-color lines):

```
B: box
define HorizShade { line right B.wid \
  from (0,-($1)*B.ht) \
  outlined rgbstring($2,$3,$4) };
ShadeObject(HorizShade, B.ht/lthick, 0,1,0,0,
1,0,0,1) at B.
```

`shadowed(box|circle|ellipse|line,` *position spec, keys*`)`

gen Object with specified shadow. *possspec* is e.g., `with .w at ...` or `at` *position*. The *keys* are `attrib=`*object attributes*; `shadowthick=`*expr*; (default `linethick*`)5/4), `shadowcolor=`*string*; (default `"gray"`), `shadowangle=`*expr*; (default −45) for box only: `rad=`*expr*;

`shielded('`*two-terminal element*`', L|U,` *line attributes*`)`

cct shielding in a `[ ]` box for two-terminal element. Arg2= blank (default) to enclose the element body; L for the left side with respect to drawing direction, R for right. Internal points `.Start, .End,` and `.C` are defined

`s_ht(`*name*`,`*default*`)` gen height of the most recent (or named) `s_box` (Section 14)

`SIdefaults` gen Sets `scale = 25.4` for drawing units in mm, and sets pic parameters `lineht = 12, linewid = 12, moveht = 12, movewid = 12, arcrad = 6, circlerad = 6, boxht = 12, boxwid = 18, ellipseht = 12, ellipsewid = 18, dashwid = 2, arrowht = 3, arrowwid = arrowht/2,`

`sidac(`*linespec*`,`*keys*`)` cct Silicon bilateral switch (Silicon Diode for Alternating Current); keys: `size=`*expr*; `env=E|`*attributes* (envelope attributes except for size; `E`=nonblank); `symbol=`*attributes* (e.g. outlined "red"); `name=`*body name*;

`sign_(`*number*`)` gen sign function

`sinc(`*number*`)` gen the sinc(x) function

`sind(`*arg*`)` gen sine of an expression in degrees

`s_init(`*name*`)` gen initialize `s_box` string label to *name* which should be unique (Section 14)

`Sin(`*integer*`)` gen sine function, *integer* degrees

`sinusoid(`*amplitude, frequency, phase, tmin, tmax, linetype*`)`
gen draws a sinusoid over the interval $(t_{\min}, t_{\max})$; e.g., to draw a dashed sine curve, amplitude *a*, of *n* cycles of length *x* from *A*, `sinusoid(a,twopi_*n/x,-pi_/2,0,x,dashed) with .Start at A`

`sl_box(`*stem linespec, keys, stem object*`)`
SLD One-terminal SLD element: argument 1 is a *linespec* to define the stem or, in the case of a zero-length stem, one of `U, D, L, R,` or an angle in degrees, optionally followed by `at` *position*. The position is *Here* by default.
Argument 2 contains semicolon (;)-separated key-value attributes of the head: `name=`*Name* (default *Head*); `lgth=`*expr*; `wdth=`*expr*; `text="`*text*`"`, `box=`*box pic attributes*.
If argument 3 is null then a plain stem is drawn; if it is of the form `S:`*keys* or `S`*n*`:`*keys* an *n*-line slash symbol is overlaid on the stem; otherwise the keys are for an overlaid breaker, so that a `C` specifies a default closed breaker, `O` an open breaker, `X, /,` or `\` for these marks, or `sl_ttbox` key-value pairs defining box attributes for the breaker (default name *Br*)
(Section 11)

`sl_breaker(`*linespec,* `type=[A|C][D];`*ttbox args*`)`
SLD Two-terminal SLD element: type `A` (the default) is for a box breaker; type `C` for a curved breaker; adding a `D` puts drawout elements in the input and output leads. Otherwise, the arguments are as for `sl_ttbox` (Section 11)

`sl_busbar(`*linespec, np, keys*`)` SLD Composite SLD element drawn in a `[ ]` block. A busbar is essentially a thick straight line drawn along the *linespec* with positions evenly distributed along it. For example, `line right_; sl_busbar(, up_ 4.5, 5) with .P3 at Here`.
Argument 1 is a *linespec* to define the direction and length of the busbar (but not its position, since it is drawn in a `[ ]` block).
Argument 2 is the number *np* of evenly spaced positions $P1, P2, \ldots Pnp$ along the line with $P1$ and Pnp indented from the ends of the line.
Argument 3 contains semicolon (;)-separated key-value attributes of the line: `port=D` (for a dot at each port position); `line=`*pic line attributes*. `indent=`*indent distance*. (Section 11)

`sl_capacitor`(*stem linespec, keys, breaker or* `S`*n*:*slash keys*)

SLD One-terminal SLD element: argument 1 is a *linespec* to define the stem or, in the case of a zero-length stem, one of `U, D, L, R,` or an angle in degrees, optionally followed by `at` *position*. The position is *Here* by default.
Argument 2 contains semicolon (;)-separated key-value attributes of the head:
`name=`*Name* (default `Head`);
`wdth=`*expr*; (default `sl_boxlen_/2`)
`lgth=`*expr*; (default `wdth/3`)
`head=`*attributes*
`headline=`*attributes*
Argument 3 is null for no breaker in the stem, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or `sl_ttbox` key-value pairs defining box attributes for the breaker (default name *Br*) (Section 11)

`sl_ct(at`*position*`,`*keys*`,R|L|U|D|`*degrees*`)`

SLD Composite SLD element drawn in a `[ ]` block:
The keys are as follows: `type=L|N|S[n]` (default `L; S`*n* draws an *n*-line slash symbol, default 2); `N` means no stem); `scale=`*expr* `(default 1.0)`; `grnd=`*expr* `attached ground at given angle (type S or N))`; `sep=`*expr*; `stemlgth=`*expr*; `wdth=`*expr*; `direct=U|D|L|R|`*degrees* (drawing direction).
Key `stemlgth` is the length of the leads at the start, centre, and end, with labeled ends *Tstart, Tc,* and *Tend*. The `L` (default) variant also defines internal labels Internal labels *L* and *C* are included.
Key `sep` is the type-`S` separation from the head to the centre of the slash symbol.
Key `scale` allows scaling (default scale 1.0) but, with dpic, the `scaled` directive can also be used. (Section 11)

`sl_disk`(*stem linespec, keys, breaker*)

SLD One-terminal SLD element: argument 1 is a *linespec* to define the stem or, in the case of a zero-length stem, one of `U, D, L, R,` or an angle in degrees, optionally followed by `at` *position*. The position is *Here* by default.
Argument 2 contains semicolon (;)-separated key-value attributes of the head: `name=`*Name* (default *Head*); `text="`*text*`"`; `diam=`*expr*; `circle=`*circle pic attributes*.
Argument 3 is null for no breaker in the stem, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or `sl_ttbox` key-value pairs defining box attributes for the breaker (default name *Br*) (Section 11)

`sl_drawout(`*linespec, keys,* `R)` SLD

Two-terminal SLD element: argument 1 is a *linespec* as for ordinary two-terminal elements.
Argument 2 contains semicolon (;)-separated key-value body attributes:
`type=T` (for truncated leads); `lgth=`*expr*`, wdth=`*expr* (body size); `name=`*Name* (default *Body*); `line=`*pic line attributes*; (e.g., `thick 2`)
Argument 3 is `R` to reverse the direction of the drawn chevrons. (Section 11)

`sl_generator(`*stem linespec, keys, breaker*`)`

SLD

One-terminal SLD element: argument 2 is `type=AC|WT|BS|StatG|PV|Y|Delta` and, if `type=PV,` the `SL_box` keys; otherwise, the `sl_disk` body keys.
Argument 3 is null for no breaker in the stem, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or `sl_ttbox` key-value pairs defining box attributes for the breaker (default name *Br*) (Section 11)

`sl_grid(`*stem linespec, keys, breaker*`)`

SLD

One-terminal SLD element: argument 1 is a *linespec* to define the stem or, in the case of a zero-length stem, one of `U, D, L, R,` or an angle in degrees, optionally followed by `at` *position*. The position is *Here* by default.
Argument 2 contains semicolon (;)-separated key-value attributes of the head: `name=`*Name* (default *Head*); `lgth=`*expr*; `wdth=`*expr*.
Argument 3 is null for no breaker in the stem, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or `sl_ttbox` key-value pairs defining box attributes for the breaker (default name *Br*) (Section 11)

`sl_inductor(`*stem linespec, keys, breaker or* `Sn:`*slash keys*`)`

SLD

One-terminal SLD element: argument 1 is a *linespec* to define the stem or, in the case of a zero-length stem, one of `U, D, L, R,` or an angle in degrees, optionally followed by `at` *position*. The position is *Here* by default.
Argument 2 contains semicolon (;)-separated key-value attributes of the head:
`name=`*Name* (default `Head`);
`cycles=`*integer*; (default 4)
`side=L|R` (default `L`)
`headline=`*attributes*
Argument 3 is null for no breaker in the stem, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or `sl_ttbox` key-value pairs defining box attributes for the breaker (default name *Br*) (Section 11)

`sl_inverter(`*ttbox args*`)` SLD

Two-terminal SLD element: the arguments are as for `sl_ttbox` (Section 11)

`sl_lamp(`*stem linespec, keys, breaker*`)`

SLD

One-terminal SLD element: the arguments are as for `sl_disk` (Section 11)

`sl_load`(*stem linespec, keys, breaker*)

SLD — One-terminal SLD element: argument 1 is a *linespec* to define the stem or, in the case of a zero-length stem, one of `U, D, L, R,` or an angle in degrees, optionally followed by `at` *position*. The position is *Here* by default.
Argument 2 contains semicolon (;)-separated key-value attributes of the head: `name=`*Name* (default *Head*); `lgth=`*expr*; `wdth=`*expr*; `head=`*arrowhead pic attributes*.
Argument 3 is null for no breaker in the stem, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or `sl_ttbox` key-value pairs defining box attributes for the breaker (default name *Br*) (Section 11)

`sl_meterbox`(*stem linespec, keys, breaker*)

SLD — One-terminal SLD element: argument 1 is a *linespec* to define the stem or, in the case of a zero-length stem, one of `U, D, L, R,` or an angle in degrees, optionally followed by `at` *position*. The position is *Here* by default.
Argument 2 contains semicolon (;)-separated key-value attributes of the head: `name=`*Name* (default *Head*); `lgth=`*expr*; `wdth=`*expr*; `text="`*text*`"`, `box=`*box pic attributes*.
Argument 3 is null for no breaker in the stem, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or `sl_ttbox` key-value pairs defining box attributes for the breaker (default name *Br*) (Section 11)

`sl_reactor`(*stem linespec, keys, breaker keys, breaker keys*)

SLD — Two-terminal SLD element: argument 1 is a *linespec* as for ordinary two-terminal elements.
Argument 2 contains semicolon (;)-separated key-value body attributes: `name=`*Name* (default *Body*); `diam=`*expr*.
Argument 3 is null for no breaker in the input lead, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or key-value pairs as above defining breaker attributes except that the default breaker name is *BrI*.
Argument 4 defines the breaker in the output lead as for argument 3 except that the default breaker name is *BrO*.
(Section 11)

`sl_rectifier`(*ttbox args*) — SLD — Two-terminal SLD element: the arguments are as for `sl_ttbox` (Section 11)

`sl_slash(at` *position, keys,*`[`*n*`:]R|L|U|D|`*degrees*`)`

SLD — Slash symbol for SLD elements: draws *n* slashes in a `[]` block. The keys are `lines=`*line attributes*, `e.g., dotted thick` *expr*; `size=`*expr* (default `ht dimen_/3`).
(Section 11)

`sl_transformer3`(*linespec, keys, breaker keys, symbol keys*)

SLD Composite (block) SLD element: argument 1 is a *linespec* that can be used to set the direction and distance between primary terminals but not position.
Argument 2 contains semicolon (;)-separated key-value body attributes: `name=`*Name* (default *Body*); `type=S|C` (default `S`); `scale=`*expr* (body size factor, default 1.0); `direct=L|R` (default `L`) direction of the tertiary circle and terminal relative to the drawing direction; `body=`*circle attributes*.
Argument 3 is colon (:)-separated sequence of up to three breaker attribute specifications for the input, output, and teriary breaker in order. A null or blank means no breaker, `tt_breaker` specifications otherwise. Default breaker names are *BrI* and *BrO* as for `sl_transformer,` and *Br* for the third breaker.
Argument 4 is colon (:)-separated sequence of up to three symbol specifications for the input, output, and teriary circle in order. A null or blank means no symbol; `Y` for a Y-symbol; `Delta` for a Δ symbol; otherwise, other customization commands expanded in a `{}` pair.
(Section 11)

`sl_transformer`(*linespec, keys, input breaker keys, output breaker keys, input circle inner object, output circle inner object*)

SLD Two-terminal SLD element: argument 1 is a *linespec* as for ordinary two-terminal elements.
Argument 2 contains semicolon (;)-separated key-value body attributes: `name=`*Name* (default *Body*); `scale=`*expr* (body size factor, default 1.0); `type=I|S|A[R]` (default `I`).
Additional type `I` keys are `cycles=`*integer* (default 4); `core=A|M[`*n*`]|P[`*n*`]|K[`*n*`]`, *n*=*integer* (default 2 lines).
Additional type `S` keys are `body=`*circle pic attributes* e.g., `shaded "`*color*`"`.
Type `A` keys are `body=`*circle pic attributes*. Type `AR` means right orientation.
Argument 3 is null for no breaker in the input lead, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or key-value pairs as above defining breaker attributes except that the default breaker name is *BrI*.
Argument 4 defines the breaker in the output lead as for argument 3 except that the default breaker name is *BrO*.
Argumentss 5 and 6 for the input and output circles respectively are: `Y` for a Y-symbol; `YN` for a Y-symbol with ground; `Delta` for a Δ symbol; otherwise, other customization commands expanded in a `{}` pair.
(Section 11)

`sl_ttbox`(*linespec, keys, input breaker keys, output breaker keys*)
SLD Two-terminal SLD element: argument 1 is a *linespec* as for ordinary two-terminal elements.
Argument 2 contains semicolon (;)-separated key-value body attributes: `name=`*Name* (default *Body*); `lgth=`*expr*; `wdth=`*expr*; `text="`*text*`"`; `box=`*box pic attributes*; `supp=`*additional* `rotbox` *commands*.
Argument 3 is null for no breaker in the input lead, `C` for a default closed breaker, `O` for an open breaker, `X, /,` or `\` for these marks, or key-value pairs as above defining breaker attributes except that the default breaker name is *BrI*.
Argument 4 defines the breaker in the output lead as for argument 3 except that the default breaker name is *BrO*.
(Section 11)

`s_name` gen the value of the last `s_init` argument (Section 14)

`sourcerad_` cct default source radius

`slantbox`(*wid, height, x offset, y offset, attributes*)
dpictools Trapezoid formed from a box with top corners displaced right by x offset and right corners displaced up by y offset.

`source`(*linespec*`,` *char or chars*`,` *diameter*`,R,` *body attributes, body name*)
cct Source; arg2 blank or:
`AC`: AC source;
`B`: bulb;
`F`: fluorescent;
`G`: generator;
`H`: step function;
`I`: current source;
`i`: alternate current source;
`ii`: double arrow current source;
`ti`: truncated-bar alternate current source;
`dci`: DC current source;
`L`: lamp;
`N`: neon;
`NA`: neon 2;
`NB`: neon 3;
`P`: pulse;
`Q`: charge;
`R`: ramp; `S`: sinusoid;
`SC`: quarter arc, `SCr` right orientation;
`SE`: arc, `SEr` right orientation;
`T`: triangle;
`U`: square-wave;
`V`: voltage source;
`v`: alternate voltage source;
`tv`: truncated-bar alternate voltage source;
`dcv`: DC voltage source;
`X`: interior X;
other: custom interior label or waveform;
arg 4: R: reversed polarity;
arg 5 modifies the circle (body) with e.g., color or fill;
arg 6 names the body `[ ]` block (Section 4.2)

`speaker( U|D|L|R|`*degrees*`,`*size*`,H,attributes)`
cct speaker, *In1* to *In7* defined; `H`: horn (Section 6)

`sprod3(`*scalar, vec1, vec2*`)` dpictools Multiplied vector by scalar arg1: *vec2* = *vec1* * *arg1*.

`sprod3D(a,x,y,z)` 3D scalar product of triple x,y,z by arg1

`sp_` gen evaluates to medium space for gpic strings

`sqrta(`*arg*`)` gen square root of the absolute value of *arg*; i.e., `sqrt(abs(`*arg*`))`

`SQUID(`*n, diameter, initial angle*`, ccw|cw)`
cct Superconducting quantum interface device with n junctions labeled `J1, ... J`*n* placed around a circle with initial angle -90 deg (by default) with respect to the current drawing direction. The default diameter is `dimen_`

`s_` gen .s with respect to current direction

`stackargs_('`*stackname*`',`*args*`)`
gen Stack arg 2, arg 3, ... onto the named stack up to a blank arg

`stackcopy_('`*name 1*`','`*name 2*`')`
gen Copy stack 1 into stack 2, preserving the order of pushed elements

`stackdo_('`*stackname*`',`*commands*`)`
gen Empty the stack to the first blank entry, performing arg 2

`stackexec_('`*name 1*`','`*name 2*`',`*commands*`)`
gen Copy stack 1 into stack 2, performing arg3 for each nonblank entry

`stackprint_('`*stack name*`')` gen Print the contents of the stack to the terminal

`stackreverse_('`*stack name*`')` gen Reverse the order of elements in a stack, preserving the name

`stacksplit_('`*stack name*`',`*string*`,`*separator*`)`
gen Stack the fields of *string* left to right separated by nonblank *separator* (default .). White space preceding the fields is ignored.

`sum3(`*vec1, vec2, vec3*`)` dpictools The 3-vector sum *vec3* = *vec1* + *vec2*.

`sum3D(`*x1,y1,z1,x2,y2,z2*`)` 3D sum of two triples

`sum_(`*a*`,`*b*`)` gen binary sum

`sus(`*linespec, chars, label*`)` cct Wrapper to place an SUS thyristor as a two-terminal element with `[ ]` block label given by the third argument (Section 6.1)

`svec_(`*x*`,`*y*`)` log scaled and rotated grid coordinate vector

`s_wd(`*name*`,`*default*`)` gen width of the most recent (or named) `s_box` (Section 14)

`switch(`*linespec*`,L|R,[C|O][D],[B|D])`
cct SPST switch (wrapper for bswitch, lswitch, and dswitch), arg2: `R`: right orientation (default `L` for left);
if arg4=blank (knife switch): arg3 = `[O|C][D][A]`, `O`: opening, `C`: closing, `D`:dots, `A`: blade arrowhead;
if arg4=B (button switch): arg3 = `O|C`: `O`: normally open, `C`: normally closed;
if arg4=`D`: arg3 = same as for dswitch (Section 4.2)

`sw_` gen .sw with respect to current direction

T

`tapped('`*two-terminal element*`', [`*arrowhd* `| type=`*arrowhd*`;name=`*Name*`],` *fraction, length, fraction, length,* ⋯`)`
cct Draw the two-terminal element with taps in a [ ] block (see `addtaps`). *arrowhd* = blank or one of `. - <- -> <->`. Each fraction determines the position along the element body of the tap. A negative length draws the tap to the right of the current direction; positive length to the left. Tap names are Tap1, Tap2, ⋯ by default or Name1, Name2, ⋯ if specified. Internal block names are `.Start`, `.End`, and .C corresponding to the drawn element, and the tap names (Section 6)

`ta_xy(`*x, y*`)` cct macro-internal coordinates adjusted for `L|R`

`tbox(`*text,wid,ht*`,<|>|<>,`*attributes*`, U|D|L|R|`*degrees*`)`
cct Pointed terminal box in a [ ] block, drawn in the current direction unless specified in arg6. The *text* is placed at the rectangular center in math mode unless the text begins with `"` or `sprintf` in which case the argument is used literally. Arg 4 determines whether the point is forward, backward, or both with respect to the current drawing direction. (Section 6)

`tconn(`*linespec, chars*`|`*keys*`,` *wid*`)`
cct Terminal connector drawn on a linespec, with head enclosed in a `[]`block. The permissible *chars* are: `> | >> | < | << | A | AA | M | O | OF`. Type `O` draws a node (circle); `OF` a filled circle. Type `M` is a black bar; `A` is an open arc end; type `AA` a double open arc. Type `>` (the default) is an arrow-like output connector; `<` and ≪ input connectors. Arg 3 is arrowhead width or circle diameter when key-value pairs are not used. If keys are specified, they are `type=`*chars* as previously; `wdth=`*expr*; `lgth=`*expr*; `sep=`*expr*; `head=`*attributes except* `lgth, wdth.` The key `sep=` is the double-head separation (Section 6)

`testexpr(`*variable, expr1, expr2, ...*`)`
dpictools Set the variable given by arg1 to the index of the first true alternative in a sequence of logical expressions, e.g., `testexpr(i, 1>2, 1<2 )` sets *i* to 2. The variable is set to 0 if no test is true.

`tgate(`*linespec,* `[B][R|L])` cct transmission gate, `B=` ebox type; `L=` oriented left (Section 6.1)

`thermocouple(`*linespec, wid, ht,* `L|R [T])`
cct Thermocouple drawn to the left (by default) of the *linespec* line. A `T` argument truncates the leads so only the two branches appear. `R=` right orientation. (Section 4.2)

`thicklines_(`*number*`)` gen set line thickness in points

`thinlines_(`*number*`)` gen set line thickness in points

`threeD_init` 3D initialize 3D transformations (reads `lib3D.m4`)

`thyristor(`*linespec*`,[SCR|SCS|SUS|SBS|IEC][`*chars*`])`
cct Composite thyristor element in `[]`block, types:
`SCR`: silicon controlled rectifier (default),
`SCS`: silicon controlled switch,
`SUS`: silicon unilateral switch,
`SBS`: silicon bilateral switch,
`IEC`: type IEC.
Chars to modify or define the element:
`K`: open arrowheads,
`A`: arrowhead,
`F`: half arrowhead,
`B`: bidirectional diode,
`E`: adds envelope,
`H`: perpendicular gate (endpoint *G*),
`N`: anode gate (endpoint *Ga*),
`U`: centre line in diodes,
`V`: perpendicular gate across arrowhead centre,
`R`: right orientation,
`E`: envelope (Section 6.1)

`thyristor_t(`*linespec, chars, label*`)`
cct Wrapper to place a thyristor as a two-terminal element with `[ ]` block label given by the third argument (Section 6.1)

`tikznode(`*Tikz node name, position*`)`
pgf insert Ti*k*z code to define a zero-size Ti*k*z node at *location*(default `Here`) to assist with inclusion of pic code output in Ti*k*z diagrams. This macro must be invoked in the outermost pic scope. (Section 15.1)

`tline(`*linespec*`,`*wid*`,`*ht*`)` cct transmission line, manhattan direction (Section 4.2)

`ToPos(`*position*`, U|D|L|R|`*degrees*`,` *length*`)`
gen Evaluates to `from` *position* `- Rect_(`*length, angle*`) to` *position* from the polar-coordinate data in the arguments

`transformer(`*linespec*`,L|R,`*np*`,[A|P][W|L][D1|D2|D12|D21],`*ns*`)`
cct 2-winding transformer or choke with terminals *P1, P2, TP, S1, S2, TS*:
arg2: `L`: left, `R`: right,
arg3: np primary arcs,
arg5: ns secondary arcs,
arg4: `A`: air core,
`P`: powder (dashed) core,
`W`: wide windings,
`L`: looped windings,
`D1`: phase dots at *P1* and *S1* end;
`D2`: at *P2* and *S2* end;
`D12`: at *P1* and *S2* end;
`D21` at *P2* and *S1* end (Section 6)

`tr_xy_init(`*origin, unit size, sign* `)`
cct initialize `tr_xy`

`tr_xy(`*x, y*`)` cct relative macro internal coordinates adjusted for `L|R`

`tstrip(R|L|U|D|`*degrees, nterms, chars*`)`
cct terminal strip, chars: I (invisible terminals), C (default circle terminals), D (dot terminals), O (omitted separator lines), `wid=`value`;` total strip width, `ht=`value`;` strip height, `box=`*shaded etc.*`;` (Section 6)

`ttmotor(`*linespec, string, diameter, brushwid, brushht*`)`
cct motor with label (Section 4.2)

`twopi_` gen 2π

U

`ujt(`*linespec*`,R,P,E)` cct unijunction transistor, right, P-channel, envelope (Section 6.1)

`unit3D(x,y,z)` 3D unit triple in the direction of triple x,y,z

`up__` gen up with respect to current direction

`up_` gen set current direction up (Section 5)

V

`variable(‘`*element*`’, chars, [+|-]`*angle*`, length, at position)`
cct — Overlaid arrow or line to indicate variable 2-terminal element: The *chars* are
`A`: arrow,
`P`: preset,
`L`: linear,
`N`: nonlinear,
`NN`: symmetric nonlinear,
`C`: continuous,
`S`: setpwise;
`u` changes the nonlinearity direction. The angle is absolute but preceding it with a sign makes the angle (often -30 or -45) relative to the element drawing direction.
If arg5 is blank the symbol is placed over the last `[]` block (Section 4.2)

`Vcoords_(`*position*`)` — gen — The x, y coordinate pair of the position

`Vdiff_(`*position*`,`*position*`)` — gen — `Vdiff_(A,B)` evaluates to `A-(B)` with dpic, `A-(B.x,B.y)` with gpic

`vec_(`x`,`y`)` — gen — position rotated with respect to current direction

`vec_r(`x`,`y`)` — gen — Robust position rotated with respect to current direction for use in dpic loops

`vec3(`*vector*`)` — dpictools — Expands to the threee components of the vector argument separated by commas.

`View3D` — 3D — The view vector (triple) defined by `setview(`*azimuth, elevation, rotation*`)`. The `project` macro projects onto the plane through `(0,0)` and orthogonal to this vector.

`vlength(`x`,`y`)` — gen — vector length $\sqrt{x^2 + y^2}$

`vperp(`*linear object*`)` — gen — unit-vector pair CCW-perpendicular to linear object

`Vperp(`*position name, position name*`)` — gen — unit-vector pair CCW-perpendicular to line joining two named positions

`vrot_(`x`,`y`,`*xcosine*`,`*ycosine*`)` — gen — rotation operator

`vscal_(`*number*`,`x`,`y`)` — gen — vector scale operator

`Vsprod_(`*position, expression*`)` — gen — The vector in arg 1 multiplied by the scalar in arg 2

`Vsum_(`*position*`,`*position*`)` — gen — `Vsum_(A,B)` evaluates to `A+B` with dpic, `A+(B.x,B.y)` with gpic

W

`while_(‘`*test*`’,‘`*actions*`’)` — gen — Integer m4 while loop

`wid_` — gen — width with respect to current direction

`winding(L|R,` *winding diam, pitch, nturns, core wid,* `"`*core color*`")`
cct core winding drawn in the current direction; `R`: right-handed The complete spline is drawn in the current drawing direction, then parts of it are overwritten with the background core color (default white). Arg 1 contains R for right-handed winding. (Section 6)

`w_` gen .w with respect to current direction

X

`XOR_gate(`*n*`,N)` log 'xor' gate, 2 or *n* inputs; `N`: negated input. Otherwise, arg1 can be a sequence of letters `P|N` to define normal or negated inputs. The default output conforms to current ANSI standard by drawing short lines from the inputs to the gate main body. Original behaviour to omit them can be set by `define('XOR_off',-1)` either globally or for individual gates. (Section 9)

`XOR_off` log XOR and NXOR offset parameter of the input face, equal to 1 to conform to current ANSI standard by drawing short lines from the inputs to the gate main body. Original behaviour to omit them can be set by `define('XOR_off',-1)` either globally or for individual gates.

`xtal(`*linespec*`,`*keys*`)` cct Quartz crystal. The *keys* are `type=N` (default) or `type=R` (round);
type `N` keys:
`lgth=`*expr* (body length);
`wdth=`*expr* (body width);
`bxwd=`*expr* (body inner box width);
`box=` box attributes (`shaded` ...);
type `R` keys:
`outerdiam=`*expr*;
`innerdiam=`*expr*;
`outer=` outer circle attributes (`dotted` ...);
`inner=` inner circle attributes (`shaded` ...) (Section 4.2)

`xtract(`*string, substr1, substr2,* ...`)`
gen returns substrings if present

Y

`Ysymbol(at` *position, keys,* `U|D|L|R|`*degrees, attributes*`) (default U for up)`
cct Y symbol for power-system diagrams. *keys:*
`size=`*expression*`;`
`type=[N|Z][R|L]` (`N`=normal, `Z`=zig, `R`=right grounded, `L`=left grounded).
Arg4 is the attributes of the drawn line object, e.g., `outlined "red"`

Z

`zabs(`*complex value*`)` dpictools Absolute value of complex value $\sqrt{(val.x^2 + val.y^2}$

`zarg(`*complex value*`)` dpictools Angle of complex value atan2$(val.y, val.x)$

`Zcos(`*complex value*`)` dpictools Complex cosine $(\cos(val.x) * \cosh(val.y), -\sin(val.x) * \sinh(val.y))$

`Zdiff(`*complex value, complex value*`)` dpictools Complex subtraction $(val1.x - val2.x, val1.y - val2.y)$

`Zexp(`*complex value*`)` dpictools Complex exponential $((\cos(val.y), \sin(val.y)) * e^{val.x})$

`Zinv(`*complex value*`)` dpictools Complex inverse $((val.x, -val.y)/\text{zabs}(val))$

`Zprod(`*complex value, complex value*`)` dpictools Complex multiplication $(val1.x * val2.x - val1.y * val2.y, val1.y * val2.x + val1.x * val2.y)$

`Zsin(`*complex value*`)` dpictools Complex sine $(\sin(val.x) * \cosh(val.y), \cos(val.x) * \sinh(val.y))$

`Zsum(`*complex value, complex value*`)` dpictools Complex addition $(val1.x + val2.x, val1.y + val2.y)$

References

[1] J. D. Aplevich. Drawing with dpic, 2022. Dpic source distribution https://gitlab.com/aplevich/dpic.

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

[3] GNU contributors. Gnu m4 1.4.19 macro processor. *GNU*, 2021. https://www.gnu.org/software/m4/manual/m4.html.

[4] D. Girou. Présentation de PSTricks. *Cahiers GUTenberg*, 16, 1994. http://cahiers.gutenberg.eu.org/cg-bin/article/CG_1994___16_21_0.pdf.

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

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

[7] IEC. International standard database snapshot 2007-01, graphical symbols for diagrams, 2007. IEC-60617.

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

[9] B. W. Kernighan. PIC—A graphics language for typesetting, user manual. Technical Report 116, AT&T Bell Laboratories, 1991. http://doc.cat-v.org/unix/v10/10thEdMan/pic.pdf.

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

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

[12] W. Lemberg. Gpic man page, 2005. http://www.manpagez.com/man/1/groff/.

[13] O. Mas. *Pycirkuit 0.5.0*. Python Software Foundation, 2019. https://pypi.org/project/pycirkuit/.

[14] E. S. Raymond. Making pictures with GNU PIC, 1995. In GNU groff source distribution, also in the dpic package and at http://www.kohala.com/start/troff/gpic.raymond.ps.

[15] T. Rokicki. DVIPS: A TEX driver. Technical report, Stanford, 1994.

[16] F. Salvaire. Pyspice overview, 2025. https://pyspice.fabrice-salvaire.fr/releases/v1.4/overview.html.

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

[18] T. Tantau. Tikz & pgf, 2013. CTAN, http://mirrors.ctan.org/graphics/pgf/base/doc/pgfmanual.pdf.

[19] T. Thurston. Drawing with MetaPost, 2023. CTAN, https://www.ctan.org/pkg/drawing-with-metapost.

[20] T. Van Zandt. PSTricks: Postscript macros for generic tex, 2007. CTAN, http://mirrors.ctan.org/graphics/pstricks/base/doc/pst-user.pdf.