M4 Macros for Electric Circuit Diagrams in LaTeX Documents

Dwight Aplevich

Version 5.0

Contents

1 Introduction

Before every conference, I find Ph.D.s in on weekends running back and forth from their offices to the printer. It appears that people who are unable to execute pretty pictures with pen and paper find it gratifying to try with a computer[8].

This document describes a set of macros, written in the **m4** macro language[6], for producing electric circuits and other diagrams in LaTeX documents. The macros evaluate to drawing commands in **pic**, a line-drawing language[7] which is readily available and quite simple to learn. The result is a system with the advantages and disadvantages of TeX itself, since it is macro-based and non-wysiwyg, and since it uses ordinary character input. The book from which the above quotation is taken correctly points out that the payoff is in quality of diagrams, at the price of the time spent in learning how to draw them.

A collection of basic components and conventions for their internal structure are described. For particular drawings it is often convenient to create new macros, or combinations of them, using consistent conventions, so macros such as these are only a starting point. The IEEE standard[5] has been followed. The macros described here make extensive use of the characteristics of **pic** and have been designed, where possible, to be an extension of the language.

2 Using the macros

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

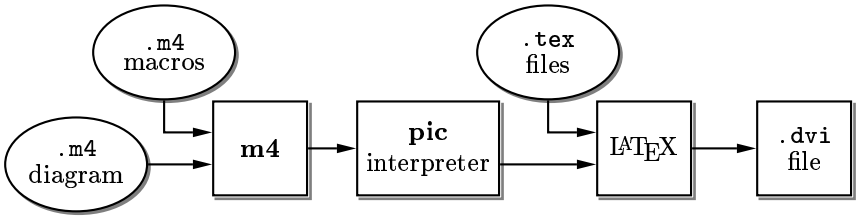


Figure 1: Inclusion of figures and macros in the LaTeX document

A file defining a diagram, together with predefined macros, is processed by **m4** and then by a **pic** interpreter to create a `.tex` file containing, for example, **tpic** specials, LaTeX graphics, (or other graphics commands, such as for **mfpic**[9] or **PSTricks**[12]), which LaTeX will convert or include in a `.dvi` file. The `.dvi` file is then to be viewed or printed by a driver capable of interpreting any `\special` commands inserted by the **pic** processor.

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

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

2.1 Quick start

The contents of file `quick.m4` are shown below, to give you enough information to produce basic labelled circuits such as Figure 2.

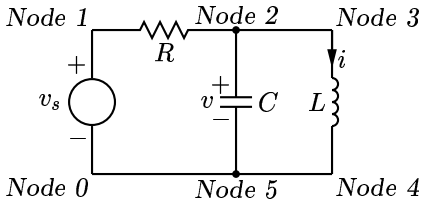


Figure 2: Diagram produced by file `quick.m4`

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

#   First define the locations of the circuit nodes and corners.
  Node0: (0,0)                  #   Absolute coordinates, in inches.
  Node1: (0,.75)

  Node2: dot(at Node1+(.75,0)) #   A dot 0.75 to the right of Node1
  Node3: Node2+(0.5,0)          #   Location 0.5 to the right of Node2
  Node4: (Node3,Node0)          #   Location (Node3.x,Node0.y)
  Node5: dot(at (Node2,Node0))

#   Draw the elements from node to node, with labels:
         source(up_ from Node0 to Node1);       llabel(-,v_s,+)
         resistor(right_ from Node1 to Node2);  rlabel(,R,)
         capacitor(down_ from Node2 to Node5);  rlabel(+,v,-); llabel(,C,)
         inductor(down_ from Node3 to Node4);   rlabel(,L,);   b_current(i)

#   Add lines as necessary:
         line from Node2 to Node3
         line from Node4 to Node0

#   Omit these if node labels are not required:
    "\sl Node 0" at Node0 below rjust
    "\sl Node 1" at Node1 above rjust
    "\sl Node 2" at Node2 above
    "\sl Node 3" at Node3 above ljust
    "\sl Node 4" at Node4 below ljust
    "\sl Node 5" at Node5 below
.PE                             # Pic input ends
```

To process the file, make sure that the libraries **libcct.m4** and **libgen.m4** are accessible. Here it is assumed that they are in your `lib` directory. Verify that **m4** is installed.

Now there are at least two possibilities. If you are using **gpic**, do the following. Type

```
m4 ~/lib/libcct.m4 quick.m4 > quick.pic
gpic -t quick.pic > quick.tex
```

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

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

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

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

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

and the document should have the statement `\usepackage{pstricks}` in the header. When many diagrams are to be processed, then a facility such as Unix **make**, which is also available in several PC versions, can be used to automate the manual commands given above.

The figure inclusion statements are

```
\begin{figure}[hbt]
   \centering
```

```
   \input quick
   \caption{Customized caption for the figure}
   \label{Symbolic label}
\end{figure}
```

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

Defining absolute node locations, as done in `quick.m4`, is simple and illustrative of some **pic** constructs, but more sophisticated construction of the above diagram would use dimensions relative to the circuit macro `dimen_` or the **pic** variable `linewid`, as described in the following sections.

3 Pic essentials

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

The visible objects are conveniently divided into two kinds, the *linear,* or line-like objects `line, arrow, spline, arc,` and the *planar* objects `box, circle, ellipse`.

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

Finally, text strings, typically meant to be typeset by LaTeX, have the double quote character at the beginning and end. Their typeset dimensions are unknown in advance, but can be obtained as demonstrated in Section 9.

3.1 Manuals

At the time of writing, the classic **pic** manual[7] can be obtained from URL:

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

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

`ftp://ece.uwaterloo.ca/pub/dpic/dpic/`

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

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

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

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

A line can be drawn as follows:

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

where *position* is defined below, or

`line` *direction distance*

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

```
scale = 25.4
```

has the effect of changing the scale to millimetres, see Section 8.

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

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

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

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

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

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

A *position* can be defined by a coordinate pair, e.g. `3,2.5`, more generally using parentheses as (*expression, expression*), and finally, using the construction (*position, position*), which takes 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)`. The current position `Here` is always defined. The coordinates of a position are accessible, e.g. `Top.x` and `Top.y` can be used in expressions. The center, start, and end of linear objects are valid positions, for example:

```
line from last line.start to 2nd last arrow.start
```

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

```
Bus23:  line up right
```

after which the object can be referenced by its symbolic name, for example:

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

To draw an arc, specify its rotation, starting point, end point, and center, but if any of these are omitted, sensible defaults are assumed.

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

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

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

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

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

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

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

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

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

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

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

```
box dashed fill
```

produces a gray dashed box.

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

```
command "string"
```

where *string* is the line to be output.

3.4 Compound objects

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

```
Ands:  [ right_
   And1:  AND_gate
```

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

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

3.5 Other language elements

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

Another possibility for defining positions is

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

which is abbreviated as

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

but care has to be used in giving this construction to **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, which must begin with a lower-case letter, and of which the scope 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 variables such as `scale`, named variables, horizontal or vertical coordinates, using the constructs *position*`.x` or *position*`.y`, dimensions of **pic** objects, e.g. `last circle.rad`.

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

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

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

the looping facility,

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

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

4 Basic two-terminal elements

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

First, all drawing macros have default arguments, so that only arguments different from default need be specified. The arguments are given in Section 13.

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

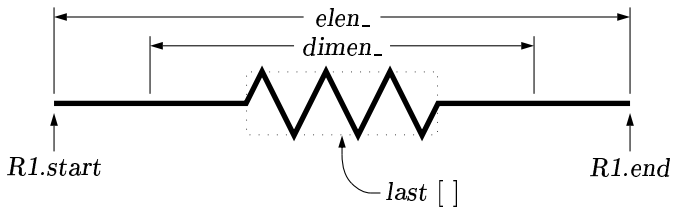


Figure 3: Resistor (with label $R1$), showing the enclosing block

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

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

R1: resistor
```

These lines and the remaining source lines are explained below:

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

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

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

4. The sequence of drawing commands to which a two-terminal macro is expanded begins with the command '`line invis` *linespec*' where *linespec* is the first argument of the macro if it is non-blank, otherwise by default the line is drawn a distance `elen_` in the current direction, which is to the right by default. The invisible line is first drawn, and the element is drawn on top of the line. The element—rather the initially-drawn invisible line—can be given a name (`R1` in the example, so that positions `R1.start` and `R1.end` are defined as shown and `R1.center` is also defined). **Pic** place names and object names begin with upper-case letters, whereas variable names begin with lower-case letters.

5. The element body is enclosed by a block, which later can be referenced for placing labels around the element. The block corresponds to an invisible rectangle of which the top and bottom are horizontal, and the sides of which are vertical, regardless of the direction in which the element is drawn. In the diagram a dotted box has been drawn to show the block boundaries.

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

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

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

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

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

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

- The line thickness is set to the default thin value of 0.4pt, and the box displaying the element body block is drawn. Notice how the width and height can be specified, and the box centre positioned at the centre of the block.
- The next paragraph draws two objects, a spline with an arrowhead, and a string left justified at the end of the spline. Other string-positioning modifiers than `ljust` are `rjust, above,` and `below.` Lines to be read by **pic** can be continued by putting a backslash as the rightmost character. Furthermore any position, such as `R1.c` in the example, has an x-coordinate and y-coordinate usable in expressions as, for example, `R1.c.x` and `R1.c.y`.
- The last paragraph invokes a macro for dimensioning diagrams. Lines can be broken before a macro argument because **m4** ignores white space before arguments.

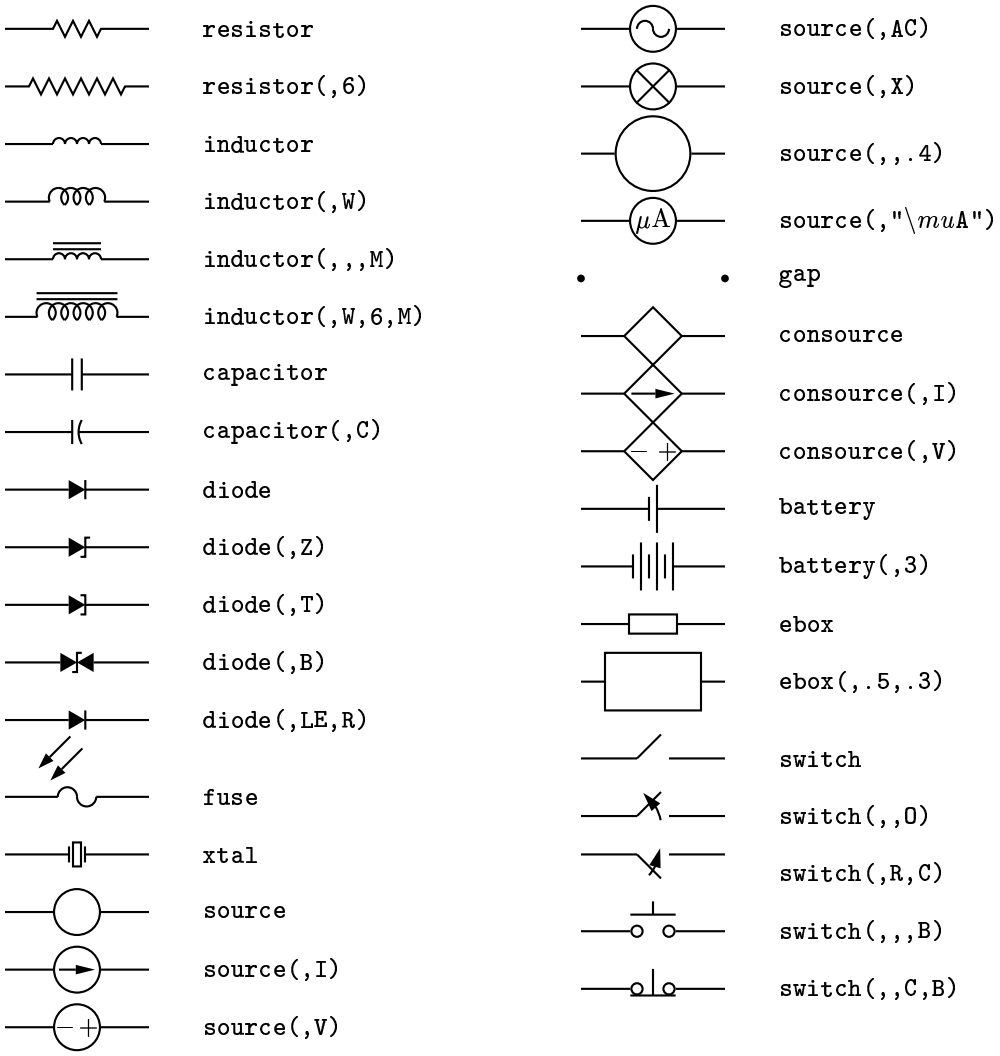


Figure 4: Two-terminal element macros, with some variations

4.1 Macro arguments

Figs. 4 and 5 are tables of the two-terminal elements included with this package, all drawn with the current direction to the right, which is the default **pic** drawing direction. Some elements are included more than once to illustrate some of their arguments. The arguments are given for each element in Section 13. In the **m4** language, macro arguments are included within parentheses

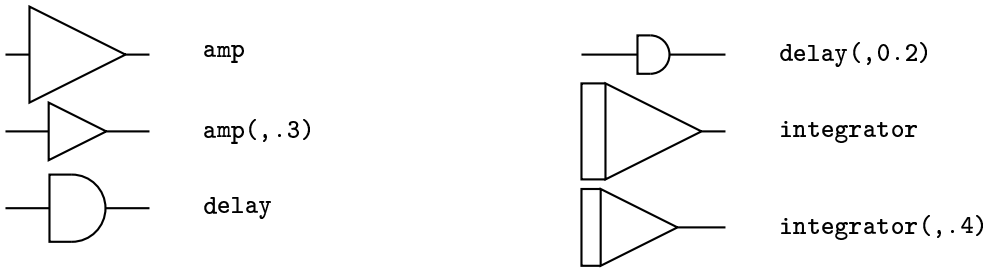


Figure 5: Amplifier, delay, and integrator

following the macro name, without spaces between the name and the opening parenthesis.

The first argument of the two-terminal elements, if included, defines the line direction and length along which the element is drawn. Other arguments of several macros are used to 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 name `up_` is a macro that resets the current macro and **pic** directional parameters to point up.

4.2 Branch-current arrows

The macro

`b_current(`*label,* `above_|below_, In|Out, Start|End)`

draws an arrow from the start of the last-drawn two-terminal element toward the body. If the fourth argument is `End`, the arrow is drawn from the end of the element toward the body. If the third element is `Out`, the arrow is drawn out, away from the body. The first argument is an optional

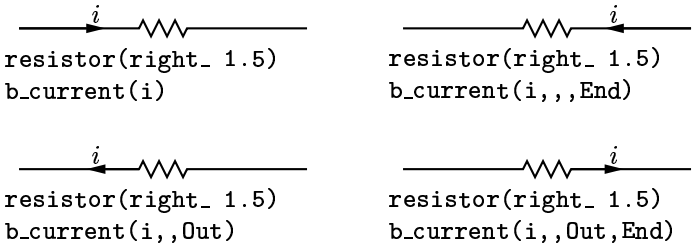


Figure 6: Illustrating `b_current`

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

4.3 Labels

Macros for labelling two-terminal elements are included:

`llabel(` *arg1,arg2,arg3* `)`
`clabel(` *arg1,arg2,arg3* `)`
`rlabel(` *arg1,arg2,arg3* `)`
`dlabel(` *long,lat,arg1,arg2,arg3* `)`

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

```
.PS
```

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

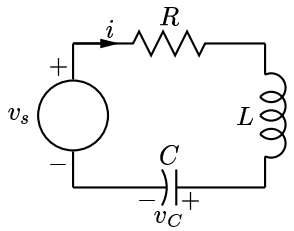


Figure 7: A loop with labelled elements

which produces Fig. 7. The macro `dlabel` performs the above functions for an obliquely-drawn element, placing the three macro arguments at `vec_(-long,lat)`, `vec_(0,lat)`, and `vec_(long,lat)` respectively relative to the centre of the element.

5 Other circuit elements

Some basic elements are not two-terminal. Fig. 8 shows a ground symbol with and without a stem, an operational amplifier, and a simple transformer. The ground symbol macro has two arguments:

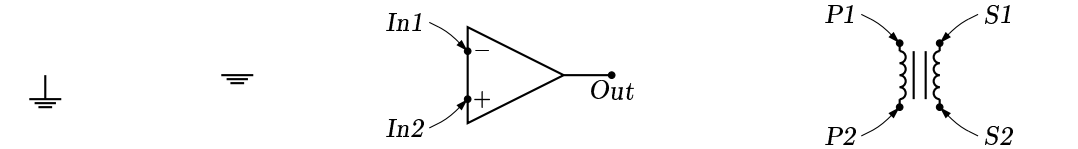


Figure 8: Miscellaneous basic elements

`ground( at` *position*`, T )`

so that, for example, the lines

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

have identical effect. Setting the second argument truncates the stem.

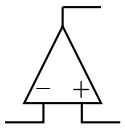
The opamp, transformer, and most other non-two-terminal elements are enclosed in a block, and therefore may be named. They generally contain named locations in the interior. An invisible line determining length and direction can be optionally specified by the first argument, as for the two-terminal elements. Instead of positioning by the first line, the enclosing block must be positioned thus: *element*`(at` *position*`)`, or using its compass corners, thus: *element* `with` *corner* `at` *position*, or, when the block contains a predefined location, thus: *element* `with` *location* `at` *position*.

The operational amplifier, with macro

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

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

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



The transformer macro is

`transformer(` *linespec*, `L|R, n_P )`

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

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

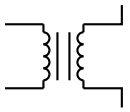


Fig. 9 shows the variations of bipolar transistor macro

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

which contains predefined internal locations *E*, *B*, *C*. The first argument defines the distance and direction from *E* to *C*, with location determined by the enclosing block as for other elements, and the

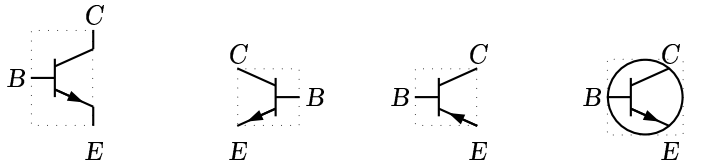
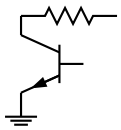


Figure 9: Bipolar transistor variants (current direction upward)

base placed to the left or right of the current drawing direction according to the second argument. Setting the third argument to 'P' creates a PNP device instead of NPN, and setting the fourth to 'E' draws an envelope around the device. Thus for example, the following code fragment places a bipolar transistor and connects a ground to the emitter and a resistor to the collector as shown:

```
up_
Q1: bi_tr(,R) with .B at (.25,.25)
ground(at Q1.E)
line up .1 from Q1.C; resistor(right_ dimen_)
```



Some FETs with predefined internal locations *S, D,* and *G* are also included, with similar arguments to those of `bi_tr,` as shown in Fig. 10. The number of possible semiconductor symbols is very large, so these macros must be regarded as prototypes.

Some other non-two-terminal macros are `dot` which has an optional argument '`at` *location*', the line-thickness macros, the `fill_` macro, and `crossover,` which is useful to show non-touching conductor crossovers, as in Figure 11.

6 Directions

Aside from its block-structure capabilities, looping, and macros, **pic** has a very useful concept of the current point and current direction, the latter unfortunately limited to `up, down, left, right.` Objects can be drawn at absolute locations or placed relative to previously-drawn objects. These `.m4` macros need to know the current direction so whenever `up, down, left, right` are used they should be written as `up_, down_, left_, right_` which are macros.

To draw circuit objects in other than the standard four directions, the macros `Point_(`*degrees*`),` `point_(`*radians*`),` and `rpoint_(`*rel linespec*`)` re-define the direction-cosine macros `x_, y_` which are

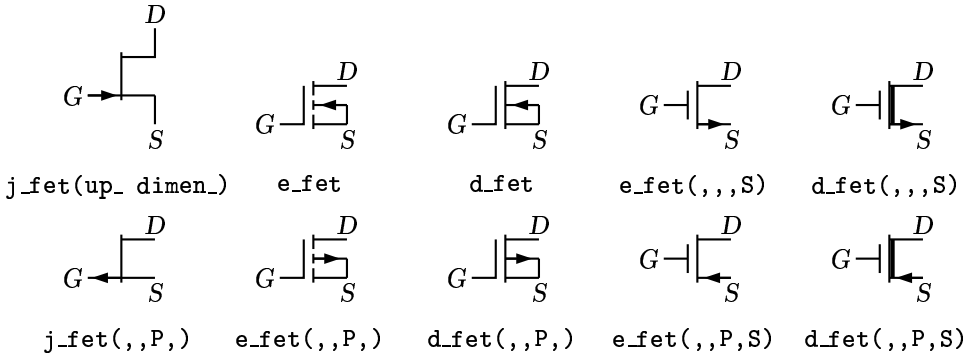


Figure 10: JFET, insulated-gate enhancement and depletion MOSFETS, and simplified versions of the MOSFETS, as in A. S. Sedra and K. C. Smith, *Microelectronic Circuits,* Oxford University Press. See also the `mosfet` and `smosfet` macros.

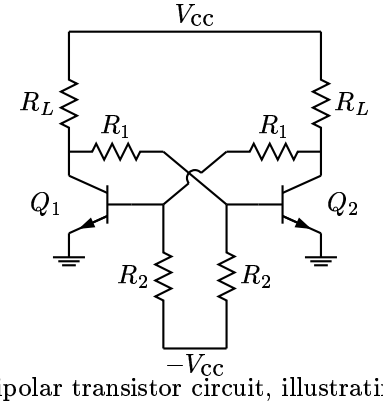


Figure 11: Bipolar transistor circuit, illustrating `crossover`

used within element-drawing macros. Thus '`Point_(-30); resistor`' draws a resistor along a line with slope -30 degrees. '`rpoint_(to Z)`' sets the current direction cosines to point to location Z. Macro `vec_(x,y)` evaluates to the position `(x,y)` rotated by the argument of the previous `Point_`, `point_` or `rpoint_` command. The macro `rvec_(x,y)` evaluates to position `Here + vec_(x,y)` and is the principal device used to define relative locations in the circuit macros. Thus, `line to rvec_(x,0)` draws a line of length `x` in the current direction.

Fig. 12 shows a circuit drawn using the above macros. The source for the figure is below, and illustrates that some hand-placement of labels using `dlabel` may be useful when elements are drawn other than horizontal or vertically.

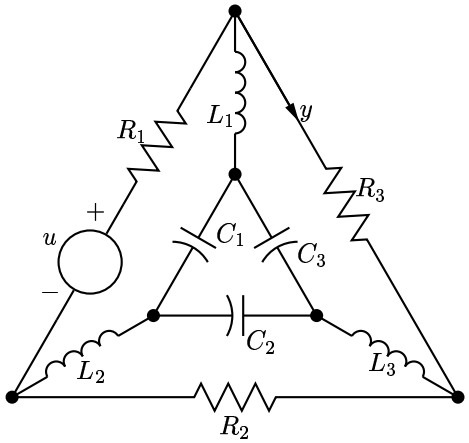


Figure 12: Illustrating elements drawn at oblique angles

```
% ex02.m4
```

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

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

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

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

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

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

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

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

The above source also illustrates that since **m4** macro arguments are separated by commas, any commas that are integral parts of the arguments must be protected either by parentheses as in `inductor(from Cr to Cr+vec_(elen_,0))`, by avoiding commas as in writing `0.5 between L and T` instead of `0.5<L,T>`, or by multiple single quotes, ‘‘,’’, as necessary.

7 Logic gates

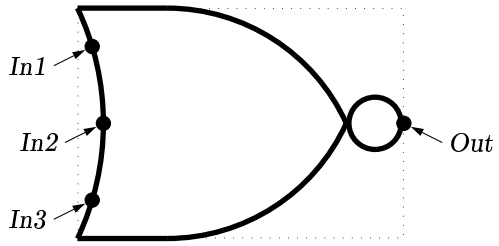


Figure 13: Illustrating `NOR_gate(3)`

Fig. 13 shows a 3-input NOR gate, with the enclosing block and predefined internal locations `Out, In1`–`In3`. Gate macros have an optional argument, an integer from 0 to N where N can be up to approximately 5, defining locations `In1,` $\cdots$ `In`N. By default $N = 2$, except for macros `NOT_gate` and `BUFFER_gate` which have one input `In1`, unless they are given an argument, which is treated as the line specification of a two-terminal element. Fig. 14 shows these and the other basic logic gates included in library `liblog.m4`. These gates are typically not two-terminal elements

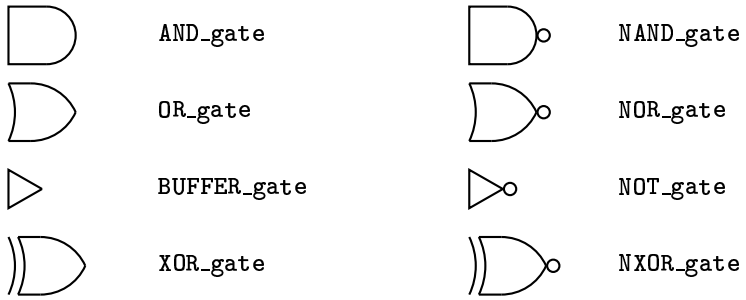


Figure 14: Basic logic gate library

and are normally drawn horizontally or vertically (although arbitrary directions may be set with e.g. `Point_(`*degrees*`)`). Each gate is contained in a block of typical height `6*L_unit` where `L_unit` is a macro intended to establish line separation for an imaginary grid on which the elements are superimposed.

The following source produces the SR flip-flop shown in Fig. 15 to illustrate change of direction. Note that when a gate is rotated, its input locations retain their positions relative to the gate body.

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

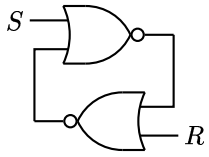


Figure 15: SR flip-flop

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

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

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

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

  `N_diam`: the diameter of NOT circles

A useful exercise might be to reproduce the example of Fig. 16.

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

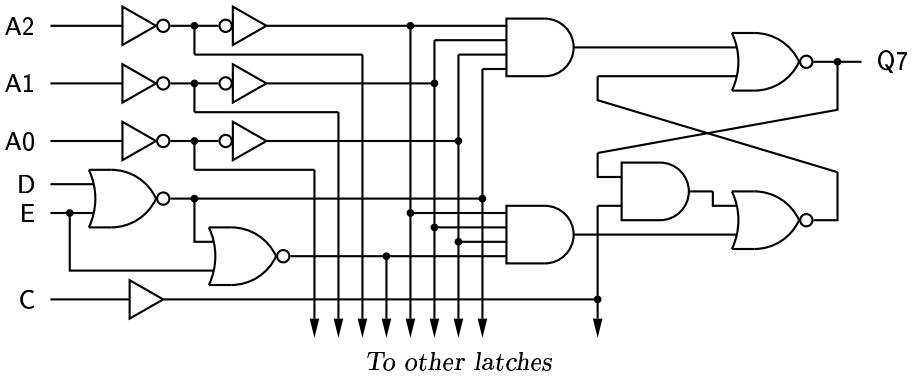


Figure 16: Multifunction latch circuit

8 Element and diagram scaling

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

First, **pic** scaling facilities will be treated, then the scaling of the defined circuit elements will be described.

8.1 Pic scaling

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

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

   The `.PS` line can be used to scale the entire drawing, regardless of its interior. Thus, for example, the line `.PS 10/25.4` scales the entire drawing to a width of 10mm.

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

2. The finished size of typeset text is independent of **pic** variables, but can be determined as in Section 9. Thus, once dimensions x and y are known, then `"text" wid` x `ht` y assigns the dimensions of `text.`

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

8.2 Circuit scaling

Scaling can be used to change the complete diagram, to change the size of elements within it, or to change unit systems, as follows:

1. The circuit elements all have default dimensions which are multiples of the **pic** variable 'linewid,' so setting this variable resets default element dimensions. The scope of a **pic** variable is the current block; therefore a sequence such as

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

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

   ```
   define(‘dimen_’,dimen_*1.2)
   ```

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

2. Assigning the **pic** variable 'scale' redefines this variable and all other **pic** built-in size variables. For example, the following lines modify `quick.m4` to use mm:

   ```
   .PS
   scale = 25.4                    # mm
   cct_init                        # Set defaults

   #   First define the locations of the circuit nodes and corners.
     Node0: (0,0)                  #   Absolute coordinates
     Node1: (0,19)

     Node2: dot(at Node1+(19,0))   #   A dot 19mm to the right of Node1
     Node3: Node2+(13,0)           #   Location 13mm to the right of Node2
   ...
   ```

9 Interaction with LaTeX

With a little hackery, the dimensions of typeset text can be obtained and used for calculations within the diagram that contains the text. The difficulty is that font metrics are not known until LaTeX is invoked, and the solution is to process the diagram twice, just as LaTeX generally requires files to be processed twice. First the diagram `.m4` source is processed by **m4** and a **pic** processor to make a `.tex` file, and the document source is LaTeXed to include the diagram and to write the required text dimensions into a supplementary file. Then the diagram `.m4` source and LaTeX document source are processed again.

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

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

has the same effect as

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

except that the line

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

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

The following small file, for example, produces the box shown in Figure 17:

```
.PS
sinclude(man.dim)  # The main input file is man.tex
box wid boxdim(Q,w) + 5pt__ ht boxdim(Q,v) + 5pt__ \
```

```
  "\boxdims{Q}{\large$\displaystyle\int_0^T e^{tA}dt$}"
.PE
```

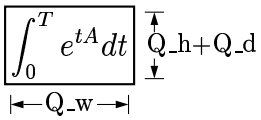


Figure 17: Fitting a box to typeset text.

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

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

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

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

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

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

assigns the correct width and height to the text strings, and the picture dimensions are then calculated correctly. Two macros are included to simplify the above example, which becomes:

```
.PS
sinclude(jobname.dim)
s_init(unique name)
B: box
  s_box(Left text) at B.w rjust
  s_box(Right text) at B.e ljust
.PE
```

The argument of `s_init` should be unique within *jobname*`.dim`. It is used to generate a unique `\boxdims` first argument for each invocation of `s_box` in the current file. If `s_init` has been omitted, the symbols "!!" are inserted into the text.

10 PSTricks tricks

This section applies only to a **pic** processor (**dpic**) that creates PSTricks output and that is capable of passing lines beginning with '\' directly to the output. Arbitrary PSTricks commands can be mixed with **m4** input to create complicated effects, but some effects are both commonly required and simple.

The rotation of text is illustrated by the file

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

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

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

```
\pscustom[linecolor=black,linewidth=0.4pt,fillstyle=solid,fillcolor=black]{
drawing commands for an arbitrary closed curve
\relax}
```

In the above example, two lines beginning with \ are passed through to the output. The first invokes the **PSTricks** `pscustom` command, with line, fill, and color options. The second line serves to insert the required closing brace, but could be replaced by the line

```
command "}%"
```

instead. The intermediate lines above define the closed curve to be filled. 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

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

in the drawing commands, as is done in `pstricks.m4`.

11 Developer's notes

Because **gpic** or its equivalent were not then available, several years ago in the course of writing a book I took a few days off to write a **pic**-like interpreter **(dpic)** that produced latex picture objects. More recently the interpreter has been upgraded to generate **mfpic** or **PSTricks** commands, the latter my preference because of the quality and flexibility of resulting graphics, including facilities for colour and rotations. In addition, **xfig** 3.1 output has been added. I preferred the more powerful **m4** macro processor to **pic** macros, and therefore **m4** is required here, although **dpic** now supports **pic**-like macros. Free versions of **m4** are available for Unix, Windows, and other operating systems. If starting over today would I not just use the **gpic** source and change its back end? Good question. Maybe. Another question might be, why not use **PSTricks** alone, or one of the other drawing packages available these days? The answer is that **pic** is a good choice for the moderate geometrical calculations that are sometimes necessary in line drawings. The language is also simple to learn, and more importantly to read, especially for backslashophobics like me. Although it is not a sophisticated programming language, **pic** has built-in looping and block-structure constructs that combine power with simplicity, and it has stood the test of time. However, no choice of tool is without compromise, and making good graphics is time-consuming no matter how it is done.

Gpic and **dpic** pass unaltered any line beginning with a \ character, allowing TeX or PSTricks macros to be invoked, for example to set parameters such as colour or fill values.

Using the **mfpic** output of **dpic** it is possible to produce Metafont alphabets of circuit elements or their subcomponents, thereby essentially removing dependence on device drivers, but with the complication of treating every graphic subcomponent as a TEX box.

The **xfig** output of **dpic** allows elements to be defined and fine-tuned, and then using the interactive graphics of **xfig** to be quickly assembled into a circuit. Further refinement of the elements might still be required.

Because the set of common circuit components and potential macros is huge, an industrial-strength system developed from the one described here would probably require a combination of all of the techniques described, including the ability to define and build character sets dynamically from macros.

12 Bugs

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

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

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

   `there is no variable ‘rp_ang’`

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

   `cct_init`

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

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

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

   when it should be

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

   This error is caused by an unfortunate inconsistency between the two-terminal elements and linear **pic** objects.

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

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

   produces an error, whereas

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

   and

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

   do not.

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

   ```
   "‘‘inverse’’"
   ```

   will be typeset as if it were

   ```
   "‘inverse’".
   ```

   The cure is to add single quotes.

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

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

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

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

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

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

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

   To help avoid name conflicts, library macros intended only for internal use have names that begin with `m4`.

   A good rule is to process diagrams in separate files, and to use the TeX `\input` command to include the result. If extensive use of strings that conflict with macro names is required, then another solution is to replace the strings by macros to be expanded by LaTeX , for example the diagram

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

   with the LaTeX macro

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

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

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

   rather than

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

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

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

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

13 List of macros

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

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

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

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

`AND_gate(`*n*`)`	log	basic 'and' gate, 2 or *n* inputs
`AND_ht`	log	height of basic 'and' and 'or' gates
`AND_wd`	log	width of basic 'and' and 'or' gates
`BUFFER_gate(`*linespec*`)`	log	basic buffer, 1 input or as a 2-terminal element
`BUF_ht`	log	basic buffer gate height
`BUF_wd`	log	basic buffer gate width
`Cos(`*integer*`)`	gen	cosine function, *integer* degrees
`E_`	gen	the constant *e*
`FlipFlop(D|T|RS|JK)`	log	experimental flip-flops
`G_hht_`	log	gate half-height
`HOMELIB_`	all	directory containing libraries
`Intersect_(`*Name1*`,`*Name2*`)`	gen	intersection of two named lines
`L_unit`	log	logic-element grid size
`NAND_gate(`*n*`)`	log	'nand' gate, 2 or *n* inputs
`NOR_gate(`*n*`)`	log	'nor' gate, 2 or *n* inputs
`NOT_gate(`*linespec*`)`	log	'not' gate, 1 input or as a 2-terminal element
`NXOR_gate(`*n*`)`	log	'nxor' gate, 2 or *n* inputs
`N_diam`	log	diameter of 'not' circles
`OR_gate(`*n*`)`	log	'or' gate, 2 or *n* inputs
`Point_(`*integer*`)`	gen	sets direction cosines in degrees
`Rect_(`*radius*`,`*angle*`)`	gen	(deg) polar-to-rectangular conversion
`Sin(`*integer*`)`	gen	sine function, *integer* degrees
`XOR_gate(`*n*`)`	log	'xor' gate, 2 or *n* inputs
`above_`	gen	string position above relative to current direction
`abs_(`*number*`)`	gen	absolute value function
`amp(`*linespec*`,`*size*`)`	cct	amplifier
`battery(`*linespec*`,n)`	cct	n-cell battery, default 1 cell
`below_`	gen	string position relative to current direction
`bi_tr(`*linespec*`,L|R,P,E)`	cct	left or right, N or P-type bipolar transistor, without or with envelope
`boxdim(`*name*`,h|w|d|v,`*default*`)`	gen	evaluate, e.g. *name*`_w` if defined, else *default* if given, else 0 `v` gives sum of `d` and `h` values
`b_current(`*label*`,`*pos*`,In|Out,Start|End)`	cct	draw and label branch-current arrow
`capacitor(`*linespec*`,C)`	cct	capacitor, straight or curved-plate
`clabel(`*label*`,`*label*`,`*label*`)`	cct	centre triple label
`consource(`*linespec*`,V|I)`	cct	voltage or current controlled source
`cosd(` *arg (degrees)*`)`	gen	cosine of a general argument in degrees
`cross(` `at` *location*`)`	gen	plots a small cross
`cross3D(x1,y1,z1,x2,y2,z2)`	3D	cross product of two triples
`crossover(`*linespec*`, L|R, Line1, ...)`	cct	line jumping left or right over named lines
`crosswd_`	gen	cross dimension
`csdim_`	cct	controlled-source width
`d_fet(`*linespec*`,L|R,P,S,E)`	cct	left or right, N or P depletion MOSFET, normal or simplified, without or with envelope
`darrow(`*linespec*`,t,t,`*width*`)`	darrow	double arrow
`dcosine3D(i,x,y,z)`	3D	extract i-th entry of triple x,y,z
`delay(`*linespec*`,`*size*`)`	cct	delay element
`delay_rad_`	cct	delay radius
`dend(` `at` *location*`)`	darrow	close (or start) double line
`diff_(`*a*`,`*b*`)`	gen	difference function
`diff3D(x1,y1,z1,x2,y2,z2)`	3D	difference of two triples
`dimen_`	cct	size parameter for circuit elements
`dimension_(`*linespec*`,`*offset*`,`*label*`,`*label wid*`,`*tic offset*`)`	gen	macro for dimensioning diagrams

diode(*linespec*,N|B|T|Z|LE,L|R)	cct	diode: normal, bi-directional, tunnel, zener, LED with arrows L, R
dlabel(*long*,*lat*,*label*,*label*,*label*)	cct	general triple label
dleft	darrow	double line left turn
dline(*linespec*,t,t,*width*)	darrow	double line
dlinewid	darrow	width of double lines
dn_	gen	sets down relative to current-direction
dot(at *location*,*radius*,*fill*)	gen	draw a (filled) dot
dot3D(x1,y1,z1,x2,y2,z2)	3D	dot product of two triples
dotrad_	gen	dot radius
down_	gen	sets current direction to down
dright	darrow	double arrow right turn
dtee('*direction*')	darrow	double arrow tee junction
dtor_	gen	degrees to radians conversion constant
e_	gen	.e relative to current direction
e_fet(*linespec*,L|R,P,S,E)	cct	left or right, N or P enhancement MOSFET, normal or simplified, without or with envelope
ebox(*linespec*,*length*,*ht*)	cct	two-terminal box element with adjustable dimensions
eleminit_(*linespec*)	cct	internal line initialization
elen_	cct	default element length
em_arrows(*linespec*,L|R,U|D)	cct	nonionizing radiation arrows
expe	gen	exponential, base e
fuse	cct	fuse symbol
Fector(x1,y1,z1,x2,y2,z2)	3D	vector with 3-dimensonal arrowhead with top face normal to x2,y2,z2, projected on current view plane
fill_(*number*)	gen	fill macro, 0=black, 1=white
gap(*linespec*,*fill*)	cct	gap with dots
glabel_	cct	internal general labeller
gpic_	gpic	defined to signify gpic is being used
grid_(*x*,*y*)	log	absolute grid location
ground(at *location*,T)	cct	ground, without stem for nonblank 2nd arg
hop(L|R,at *location*)	cct	conductor crossing another to left or right
hoprad_	cct	hop radius
ht_	gen	height relative to current direction
inductor(*linespec*,W,*n*,M)	cct	inductor, narrow or wide, 4 or n arcs, without or with magnetic core
integrator(*linespec*,*size*)	cct	integrating amplifier
intersect_(*line1*.start,*line1*.end,*line2*.start,*line2*.end)	gen	intersection of two lines
j_fet(*linespec*,L|R,P,E)	cct	left or right, N or P JFET, without or with envelope
left_	gen	left with respect to current direction
length3D(x,y,z)	3D	Euclidean length of triple x,y,z
linethick_(*number*)	gen	set line thickness in points
lin_leng(*line-reference*)	gen	calculate the length of a line
ljust_	gen	ljust with respect to current direction
llabel(*label*,*label*,*label*)	cct	triple lable on left side of the element
loc_(*x*, *y*)	gen	location adjusted for current direction
log10E_	gen	constant $\log_{10}(e)$
loge	gen	logarithm, base e
lt_	gen	left with respect to current direction
manhattan	gen	sets direction cosines for left, right, up, down
mfpic_	mfpic	defined to signify mfpic is being used
mosfet(*linespec*,L|R,P,D,E)	cct	left or right, N or P, enhancement or depletion MOSFET, without or with envelope
m4_arrow(*linespec*,*ht*,*wid*)	gen	arrow with adjustable head, filled when possible
n_	gen	.n with respect to current direction

`ne_`	gen	.ne with respect to current direction
`neg_`	gen	unary negation
`nw_`	gen	.nw with respect to current direction
`opamp(`*linespec*`,`*label*`,`*label*`,`*size*`,P)`	cct	operational amplifier with −, + or other internal labels, specified size, and optional power connections
`open_arrow(`*linespec*`,`*ht*`,`*wid*`)`	gen	arrow with adjustable open head
`point_(`*angle*`)`	gen	(radians) set direction cosines
`polar_(`*x*`,`*y*`)`	gen	rectangular-to polar conversion
`print3D(x,y,z)`	3D	write out triple for debugging
`prod_(`*a*`,`*b*`)`	gen	binary multiplication
`project(`*x*`,(`*y*`,(`*z*	3D	3D to 2D projection
`psset_(`*PSTricks settings*`)`	gen	set PSTricks parameters
`pstricks_`	pstricks	defined to signify PSTricks is being used
`pt_`	gen	big point size factor, in scaled inches, (`*scale/72`)
`pt__`	gen	TEX point size factor, in scaled inches, (`*scale/72.27`)
`rect_(`*radius*`,`*angle*`)`	gen	(radians) polar-rectangular conversion
`resistor(`*linespec*`,n)`	cct	resistor, n peaks, default 3
`right_`	gen	set current direction right
`rjust_`	gen	right justify with respect to current direction
`rlabel(`*label*`,`*label*`,`*label*`)`	cct	triple lable on right side of the element
`rot3Dx(`*radians*`,x,y,z)`	3D	rotates x,y,z about x axis
`rot3Dy(`*radians*`,x,y,z)`	3D	rotates x,y,z about y axis
`rot3Dz(`*radians*`,x,y,z)`	3D	rotates x,y,z about z axis
`rpoint_(`*angle*`)`	gen	(radians) set direction cosines
`rpos_(`*position*`)`	gen	Here + *position*
`rt_`	gen	right with respect to current direction
`rtod_`	gen	constant, degrees/radian
`rvec_(`*x*`,`*y*`)`	gen	location relative to current direction
`s_`	gen	.s with respect to current direction
`s_box(`*text*`)`	gen	generate dimensioned text string using `\boxdims` from `boxdims.sty`
`s_init(`*name*`)`	gen	initialize `s_box` string label to *name* which should be unique
`se_`	gen	.se with respect to current direction
`setview(`*azimuth degrees*`,`*elevation degrees*`)`	3D	set projection viewpoint
`shade(`*gray value*`,`*closed line specs*`)`	gen	fill arbitrary closed curve
`shadebox(`*box specification*`)`	gen	box with edge shading
`sign_(`*number*`)`	gen	sign function
`sind(` *arg (degrees)*`)`	gen	sine of a general argument in degrees
`smosfet(`*linespec*`,L|R,P,D,E)`	cct	simplified MOSFET left or right, N or P, enhancement or depletion, without or with envelope
`source(`*linespec*`,V|I|AC|X|`*string*`,`*diameter*`)`	cct	source, blank or voltage or current or AC or X or labelled
`sourcerad_`	cct	default source radius
`sprod3D(a,x,y,z)`	3D	scalar product of triple x,y,z by a
`sp_`	gen	evaluates to medium space for gpic strings
`sum_(`*a*`,`*b*`)`	gen	binary sum
`sum3D(x1,y1,z1,x2,y2,z2)`	3D	sum of two triples
`svec_(`*x*`,`*y*`)`	log	scaled and rotated grid coordinate vector
`sw_`	gen	.sw with respect to current direction
`switch(`*linespec*`,L|R,C|O,B)`	cct	SPST switch left or right, blank or closing or opening arrow, or button
`thicklines_(`*number*`)`	gen	set line thickness in points
`thinlines_(`*number*`)`	gen	set line thickness in points
`transformer(`*linespec*`,L|R,n)`	cct	2-winding transformer, left or right, n arcs
`twopi_`	gen	2π

`unit3D(x,y,z)`	3D	unit triple in the direction of triple x,y,z
`up_`	gen	set current direction up
`up__`	gen	up with respect to current direction
`vec_(`*x*`,`*y*`)`	gen	position rotated with respect to current direction
`vrot_(`*x*`,`*y*`,`*xcosine*`,`*ycosine*`)`	gen	rotation operator
`vscal_(`*number*`,`*x*`,`*y*`)`	gen	vector scale operator
`w_`	gen	.w with respect to current direction
`wid_`	gen	width with respect to current direction
`xtal(`*linespec*`)`	cct	quartz crystal

References

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

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

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

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

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

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

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

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

[9] T. Leathrum and G. Tobin. Pictures in TeX with metafont, 1996. Mfpic manual.

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

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

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