MFpic's METAFONT and METAPOST macros
grafbase.mf and grafbase.mp
Version: 0.4.0 beta, Date: 2000/08/31

Document Author: Geoffrey Tobin (G.Tobin@latrobe.edu.au)
Updates by: Daniel H. Luecking (luecking@comp.uark.edu)

• *Background*

METAFONT or METAPOST should be moderately familiar to anyone who is serious about understanding grafbase.mf or grafbase.mp.

The best reference for METAFONT programming is undoubtedly Professor Donald E. Knuth's *The* METAFONT*book*, first published by Addison-Wesley and copyrighted by the American Mathematical Society in 1986, ISBN 0-201-13445-4.

However, the file mf-revu.tex contains reminders of some facets of METAFONT.

For METAPOST specific information (as well as some basic METAFONT information), the METAPOST documentation by John Hobby is invaluable. It is contained in the file mpman.ps, available with most distributions of METAPOST.

• *Assumed Variables*

The file grafbase.mf assumes that the METAFONT variables w_, h_,xneg, xpos, yneg and ypos are all numeric and known. The values of w_ and h_ are set by the plain grafbase macro beginmfpic. By contrast, the four graph extents variables, xneg, xpos, yneg and ypos must be set externally. beginmfpic also sets the numeric variables w and h to w_ and h_, respectively. The user may use w and h (indeed, is encouraged to) but internally grafbase uses the underscored variants to guard against a user accidentally redefining the simpler ones (as the author of once did from mfpic using \mfobj{h}).

In practice, we would most often use grafbase together with mfpic.tex, in which case mfpic writes a METAFONT file that inputs grafbase and automatically sets the four extents, as well as determining w and h. The information for all six values is taken from the arguments to the \mfpic macro from mfpic.tex.

• *Preliminaries*

metapost.

grafbase uses the boolean variables METAPOST and METAFONT to record whether it thinks it's being used with METAPOST or METAFONT. Since METAFONT has no colors, it seems reasonable that the presence of blue is a reliable sign of METAPOST, so grafbase tests for blue is known.

At present, no grafbase code involves these booleans, relying on the different files to contain program specific code: if input grafbase occurs in a file, then METAFONT will input grafbase.mf and METAPOST will take grafbase.mp. Moreover, if METAPOST cannot find grafbase.mp and tries to input the other, the results will be disastrous. Perhaps in the future the two files can be merged, with differences controlled by these booleans.

The files grafbase.mf/p need the plain.mf/p macros. Usually, these will be available automatically, but if by any ill chance they are not, then grafbase will attempt to load the appropriate file.

In `grafbase.mf`, before calling `mode_setup`, it seems reasonable, given the purpose of mfpic and `grafbase.mf`, to require an explicit mode when calling METAFONT, instead of the frequently vexing, silent, default of using `proof` mode, which `plain` METAFONT falls back on when no mode is specified. So in that case, `grafbase.mf` preempts the default with an error message.

The `font_identifier` is set to `"MFpic graphics"`, to make the origin of the PK file obvious. Since there can be no generally useful coding scheme for graphics, the `font_coding_scheme` is set to `"Arbitrary"`.

It would be agreeable if `designsize` could be the same as that used for the TEX text labels in mfpic, say `10pt#` or thereabouts. Unfortunately, the limited wisdom of METAFONT limits dimensions to a small multiple of `designsize`, so `grafbase.mf` has to set `designsize` to be much larger. The value chosen (`128pt#`) is a power-of-two number of TeX points, and is large enough to avoid this limitation.

Such font related items are not available (or needed) in METAPOST, so `grafbase.mp` ignores modes, font identifiers and coding schemes.

- *Common Global Variables*

debug, grafbase; mftitle (macro); deg, radian, pi; unitlen, xscale, yscale, xneg, xpos, yneg, ypos; penwd, drawpen, hatchwd, hatchpen; shadewd; store (macro); ClipOn, ClipPath. Triangle, Square, Diamond, Circle, Star, Plus, Cross, SolidTriangle, SolidSquare, SolidDiamond, SolidCircle. curvetension, functiontension.

The `boolean` variable `debug` is present mainly for experimentation with grafbase, so that any diagnostics can be turned on or off a little more conveniently.

If the `boolean` variable `grafbase` is known, then grafbase has been loaded more than once, which is an error.

The `mftitle` macro is frequently used by mfpic, but not in grafbase itself. Its purpose is to create a title (see *The* METAFONT*book* chapter 22: Strings, page 187, paragraph 3), and to display that title as a message.

So that users can refer to angular quantities in a self-evident way, the `numeric` variable `deg` is set to `1`. Thus, instead of `45`, one may write `45 deg` or `45deg`, to make the meaning obvious. Of course `pi` is set to `3.1415926` and `radian` is set with `pi*radian = 180 deg;`.

The dimensions of the drawing are governed by seven variables: `unitlen, xscale, yscale, xneg, xpos, yneg, ypos`. The first three determine the scale of the units used in drawing, called the *graph* units, while the last four describe the nominal boundary of the drawing.

`unitlen` is provided in order to set the most convenient physical unit, since that may possibly not be one of the units provided by plain METAFONT (`bp#, cc#, cm#, dd#, in#, mm#, pc#, pt#`). Note that `unitlen` must, like all *mode-independent* units, be expressed in *sharp* units.

By contrast `xscale` and `yscale` provide `numeric` scaling factors for the *graph* units along the horizontal and vertical coordinate axes (relative to the natural orientation of the printed paper).

The nominal drawing boundary is the region that TEX will perceive as the (rectangular) boundary of the character, according to the `TFM` file for the font that METAFONT will produce (or the bounding box in the EPS files that METAPOST will produce). `xneg,` xpos, yneg, ypos are the coordinates in *graph* units of the left, right, bottom, and top edges of that boundary. Normally, the origin of the drawing *differs* from the reference point of the TFM file or the bounding box of the EPS file.

The initial values of these seven dimensioning variables are: unitlen := 1pt#, one TeX point in absolute (*sharp*) units; xscale := yscale := 7.227, so as to make the X and Y scale drawing units 1/10 inch each; while xneg, xpos, yneg, ypos are initialised to $0, 10, 0, 10$, so that both axes extend from 0 to 10 *graph* units.

However, the initial dimensions are not intended to be retained, and should be altered according to the needs of the drawing. When generating the METAFONT or METAPOST file by using TeX or LaTeX with the mfpic.tex macros, this is what the arguments of the \mfpic environment in mfpic.tex are for. To set unitlen, one uses the \mfpicunit macro of mfpic.tex.

When drawing, the pen variable drawpen is used, which is a circular pen. Its diameter, in *device* coordinates, is determined by the internal variable penwd, which grafbase initialises to 0.5pt. (See the Coordinate Conversion section below for a description of 'device coordinates'.)

For hatching, the pen variable hatchpen is used, which is a circular pen. Its diameter, in *device* coordinates, is determined by the internal variable hatchwd, which grafbase initialises to 0.5pt.

The store (f_s) f macro is for the purpose of storing a path expression f, which may be quite complex, into a path variable f_s. This is remarkably useful to mfpic.

The internal numeric variable ClipOn is zero when clipping of the current active_plane is disabled, and nonzero when such clipping is enabled.

The current clipping region is specified by the path array ClipPath. The numeric value of ClipPath equals the number of paths in the array, which are numbered from one upwards: ClipPath[1]...ClipPath[ClipPath] (*sic*).

Triangle, ..., SolidCircle are paths in the obvious shape. They have approximately the diameter 1 in device coordinates so they would normally be drawn scaled to appropriate dimensions. Despite the names, only the last four (Solid...) are closed paths. (For example, SolidTriangle is just Triangle & cycle.) They are used by doplot and plotsymbol, but are available for other uses.

The numeric curvetension is the tension used between nodes of the path created by mksmooth. It is initialized to 1 (METAFONT's default).

The numeric functiontension is a tension-like quantity used by mkcontrolledfcn.

METAPOST *Color Variables*

(Colors are only valid in the METAPOST version of grafbase.)

drawcolor, fillcolor, hatchcolor, headcolor; cyan, magenta, yellow; dvipsnam.mp. *(file)*

In METAPOST, red, green, blue, white, and black are predefined colors. grafbase.mp adds definitions for the constant colors cyan, magenta and yellow.

In addition grafbase.mp maintains the color variables drawcolor, fillcolor, hatchcolor and headcolor. drawcolor is used by shpath and safedraw, the basic curve drawing routines called by almost every function that draws a curve. Similarly, fillcolor is used for filling, hatchcolor for hatching, and headcolor for drawing arrowheads. These were chosen because these are the basic drawing operations of mfpic macros: drawing curves, filling interiors of curves, hatching interiors, and adding arrowheads to curves. All four are initialized to `black`.

Finally, the auxilliary file `dvipsnam.mp` contains definitions of all the color names defined in LaTeX's `dvipsnam.def`. It is normally input by files created by mfpic when the `metapost` option is selected, so all these colors are available to mfpic users.

• *Utility Macros*

map, maparr; textpairs; floorpair, ceilingpair, hroundpair; minpair, maxpair.

As noted in the source, `text` arguments, which are used in many of the macros in grafbase, are perilous in a high degree, because they can easily cause naming conflicts with local variables in macros that use them. This typically produces obscure error messages.

However, `text` arguments are very useful, and METAFONT has no useful alternative, as it lacks a sense of lists as a fully fledged data structure. Few language designers seem to learn from this excellent core feature of Lisp, so we can only muse on what might have been.

`map` $(m)(t)$ generates a new list from a text list t of one or more items, by applying the macro m to each item in turn. Lists are generally used in `for` preambles, so for each item i in t, $m(i)$ should normally be a single object. If $m(i)$ expands to some arbitrary METAFONT code, some symbols can cause problems (e.g., '`:`' can terminate a `for` preamble prematurely).

`maparr (proc) (`p`)` applies `proc`, which should be a *procedure* (a sequence of statements that does not return any value) with one argument, to each member of the array p[]. The items in the array may be of any types that `proc` can process.

`textpairs` (p, t) converts a text list t of `pair`s into an array p of `pair`s.

`chpair (proc) (`p`)` applies `proc` to each *part* of the `pair` p, then it returns the `pair` consisting of each of these two results: `(proc (xpart `p`), proc (ypart `p`))`. Here `proc` must be a macro with one `numeric` argument and which returns one `numeric` value. We describe such a macro by this mathematical notation: `proc` : `numeric` $\mapsto$ `numeric`.

`floorpair` (p) returns a `pair` comprising the floors (rounded-down integer parts) of the X and Y *parts* of the `pair` p.

`ceilingpair` (p) returns a `pair` comprising the ceilings (rounded-up integer parts) of the X and Y *parts* of the `pair` p.

`hroundpair` (p) returns a `pair` comprising the hrounds of (ie, the nearest integers to) the X and Y *parts* of the `pair` p.

`minpair` (t) returns a `pair` comprising the minimum X and Y *parts* of all the `pair`s in the text list t.

`maxpair` (t) returns a `pair` comprising the maximum X and Y *parts* of all the `pair`s in the text list t.

• *Coordinate Conversion*

ztr, invztr, setztr, zconv, invzconv; vconv, invvconv.

`grafbase.mf` employs three main (classes of) coordinate systems. The first describes *graph* coordinates; this is the user's coordinate system, in which graphs, function plots, the orientation of axis tic marks, and many other features are described. The graph coordinate system can be shifted, rotated, magnified, slanted, and flipped. Graph coordinates are device *independent*.

The second is determined by the *sharp* units of *The* METAFONT*book*; these represent an upright Cartesian system in which one millimetre means one millimetre — in any direction. Sharp coordinates are device *independent*. Device independent lengths are specified in sharp units: for example, `unitlen`.

The third describes the *device* coordinates. In METAFONT these are *pixel* coordinates, refering to units of the pixel grid in the bitmaps formed by `picture` variables and written into the GF font file that METAFONT writes; pixels are upright, but they may not be square, as METAFONT's modes cater for output devices (for example, various screens and printers) with differing aspect ratios — among other properties that affect how an image is rendered. Naturally, pixel coordinates are device *dependent*. The grafbase macros refer to pixel coordinates through plain METAFONT's `.t_` construct, through the `w` and `h_` quantities calculated by `beginmfpic`, and in its manipulations of METAFONT `picture` variables.

Below, we will use the term 'device coordinates' somewhat loosely, meaning units such as `pt` and `cm` (not `pt#` or `cm#`). These have already been set equal to some number of *horizontal* pixels by `mode_setup`. The user *could* calculate a number of pixels and use that also. However, the user should *not* adjust for nonsquare pixels using `aspect_ratio`. That is already done by the grafbase drawing commands.

The user can normally ignore the sharp coordinates, except for `unitlen`. As for graph or device coordinates: when a grafbase function requires parameters in *graph* coordinates simply use pure numerics, and when *device* coordinates are required, use pure numeric multiples of `pt` or `cm` or (inside a `beginmfpic` group) of `w` or `h`, which have already been established as a certain number of pixels. Or use `zconv` and `vconv` to convert from graph to device coordinates.

In METAPOST, device coordinates are the same as sharp coordinates. PostScript devices should perform the conversion from absolute lengths to pixels internally.

In METAFONT and METAPOST, the `transform` data type represents *affine transforms*, which are the familiar Euclidean transforms formed by translation (shifting), rotation (turning), mirror reflection (flipping), scaling (magnifying), and skewing (slanting). In the absence of translation, the operations performed are called *linear transforms*.

There is a significant difference between affine transformations, which are used to transform coordinates, and vector (displacement) transformations, because in general the former depend on the absolute position, whereas the latter act independently of it. Therefore we need a separate set of `transform`s for when `pair`s are interpreted as vectors.

`ztr` is a `transform` variable describing the conversion from *graph* to *device* coordinates.

`invztr` is the inverse of `ztr`.

`setztr` sets `ztr`.

`zconv` (a) returns the *device* coordinate `pair` corresponding to the *graph* coordinate `pair` a.

`invzconv` (v) is the inverse of `zconv`.

`vconv` (a) returns the *device* vector corresponding to the *graph* vector a.

`invvconv` (v) is the inverse of `vconv`.

• *Initial Setup*

active_plane. initpic.

`active_plane` is the active drawing plane, initially defined as the plain METAFONT `picture` variable `currentpicture` which is *not known* until the call to `clearit` in `beginchar`.

`initpic` calls `setztr`, initialises `active_plane`, and picks up a circular `pen` of diameter `penwd`.

• *Compatibility with older* `graphbase.mf`

fpicenv, endmfpicenv, bounds.

For compatibility with files produced for older versions of the graph base, usually named `graphbase.mf`, three definitions are provided.

`mfpicenv` and `endmfpicenv` are defined as empty definitions.

`bounds (a, b, c, d)` assigns `xneg, xpos, yneg, ypos` to `a, b, c, d` respectively.

The behavior of `grafbase.mp` with such old file has never been examined, so no compatibility is claimed.

• *Character Wrapper*

beginmfpic, endmfpic.

In `grafbase.mf`, `beginmfpic (ch)` is mostly a convenient abbreviation for the typical use of `beginchar` with `grafbase.mf`. However, to shield internal commands from users who might inadvertantly set variables `d`, `h` or `w`, it also sets `d_`, `h_` or `w_` to the same values for internal use. (These values are in device units)

In `grafbase.mp`, `beginmfpic` is similar to `beginfig`. The variables `d`, `h`, `w`, `d_`, `h_`, `w_` are set just as in the METAFONT version.

`endmfpic` functions like `endchar` or `endfig`, but is meant to match `beginmfpic`. Moreover, when boolean `clipall` is set, `endmfpic` clips the picture to be shipped out to the nominal dimensions. The METAPOST version also responds to the `truebbox` boolean: If this is `true`, the bounding box in the output EPS file will reflect the actual extent of the figure, if it is `false`, the bounding box will be forced to the nominal dimensions.

• *Extra Trigonometric and Hyperbolic Functions*

tand, cotd, secd, cscd, acos, asin, atan; sin, cos, tan, cot, sec, csc, invcos, invsin, invtan; exp, ln, cosh, sinh, tanh, acosh, asinh, atanh; Arg, zexp, cis.

`tand` (x), `cotd` (x), `secd` (x) and `cscd` (x) are the tangent, cotangent, secant and cosecant of x, where x is in degrees;

`sin` (x), `cos` (x), `tan` (x), `cot` (x), `sec` (x), and `csc` (x) are the the various trig functions, where x is in radians. That is, `cos (x)` is the same as `cosd (x*radian)`.

`acos` (x) and `asin` (x) are the arccosine and arcsine, in degrees, of x. Evidently, x must be between -1 and $+1$. `atan` (x) is the arctangent, and x is unrestricted.

`invsin`, `invsin` and `invtan` return angles in radians: `invsin (x)` is the same as `(asin (x))/radian`.

`exp` (x) is the value of e^x.

`ln` (x) is the natural logarithm of x.

`cosh` (x), `sinh` (x) and `tanh` (x) are the hyperbolic cosine, the hyperbolic sine and the hyperbolic tangent of x.

`acosh` (y), `asinh` (y) and `atanh` (y) are their inverses.

For complex variables we have: `Arg` z returns the angle in radians of the pair $z = (x, y)$. `zexp` z returns the pair $e^x(\cos y, \sin y)$ where $z = (x, y)$. This corresponding to complex exponentiation. Finally `cis` θ returns $(cos\theta, sin\theta)$.

- *Coordinate Systems and Transformations*

Coordinate Nesting

T_stack, T_push, T_pop; bcoords, ecoords.

In order to allow local coordinate systems to be nested without destructively interfering with each other, we define a stack of (affine) `transforms`.

`T_stack[]` is an array of `transforms`, which `grafbase` uses to implement its stack of local coordinate systems. Users should avoid using `T_stack` by name, as its security depends on its being used only by the `bccords` and `ecoords` macros. The same avoidance rule applies to all macros that have an underscore `_` in their name. This rule is a naming convention laid down by Knuth.

`T_push` (T) pushes `transform` T onto the stack `T_stack`.

`T_pop` (T) pops the stack `T_stack`, and stores its erstwhile top element in `transform` T.

The purpose of `bcoords` and `ecoords` is to enclose a local coordinate system. Thus compound objects can be built and manipulated.

`bcoords` preserves the value of `currenttransform`.

`ecoords` restores the value of `currenttransform` that was preserved by the most recent `bccords`.

Coordinate Changes

apply_t. xslant, yslant, zslant, xyswap, boost.

`apply_t` (T) changes the coordinate system: it replaces `ztr` by `transform` T followed by the `ztr`.

`xslant` s is the same as the METAFONT primitive `slanted` s : a point (x, y) maps onto $(x, y) + s(y, 0)$.

`yslant` s nicely complements `xslant` by mapping a point (x, y) onto $(x, y) + s(0, x)$.

`zslant` p, where $p = (u, v)$, maps (x, y) onto $(xu + yv, xv + yu)$. (This complements the METAFONT primitive `zscaled` p which maps (x, y) onto $(xu - yv, xv + yu)$.)

`xyswap` swaps the X and Y coordinates: $(x, y) \mapsto (y, x)$.

`boost` χ is the same as `zslant` $(\texttt{cosh}\chi, \texttt{sinh}\chi)$, which is the formula for a boost (the hyperbolic equivalent of a rotation) in special relativity.

Path Rotation

rotatedpath.

`rotatedpath` (p, θ) f returns the *graph* coordinate `path` formed by rotating the *graph* coordinate `path` f by θ degrees around the point described by the *graph* cooordinate `pair` p.

• *Bitmaps, Clipping and Rendering*

Picture to Picture — Bitwise Operations

mono. andto, picand; orto, picor; xorto, picxor; subto, picsub.

(None of these pixel oriented operations is available in METAPOST.)

METAFONT's `picture` variables are not pure bitmaps; each pixel is a negative, zero or positive weight. When the `shipout` primitive is called, as in `shipout` v`;` where v is a `picture` variable, the pixels of positive weight in v are interpreted as value one (1), and those of zero or negative weight are interpreted as zero (0). The `GF` font file uses run-length encoding to efficiently record the positions of the pixels of value one (1).

When METAFONT draws pixels in a `picture` variable, it does so via one of the forms of the `addto` primitive. The result of this *rendering* are pixels of various weights, as mentioned above.

In order to perform certain `picture` manipulation tasks, such as clipping to a given boundary curve, it is highly beneficial that pixels be either one or zero — in other words, `picture`s should be pure bitmaps.

The operation `mono` (v) changes `picture` v so that all positive pixels are replaced by one (1) and all negative or zero pixels by zero (0).

`andto` (v, w) replaces `picture` v by the bitwise *and* of v and w.

v `picand` w returns the `picture` formed by the bitwise *and* of v and w.

`orto` (v, w) replaces `picture` v by the bitwise *or* of v and w.

v `picor` w returns the `picture` formed by the bitwise inclusive *or* of v and w.

`xorto` (v, w) replaces `picture` v by the bitwise *exclusive or* of v and w.

v `picxor` w returns the `picture` formed by the bitwise *exclusive or* of v and w.

`subto` (v, w) replaces `picture` v by the bitwise *subtraction* of w from v.

v `picsub` w returns the `picture` formed by the *bitwise subtraction* of w from v. *Note:* $0 - 1 = 0$ here, as in Boolean algebra.

Color operations

(None of these color operations is available in METAFONT.)

rgb, RGB, cmyk, gray, named.

`rgb` (r, g, b) is basically a no-op. It takes three numeric parameters r, g and b, and after a little error checking, returns the color (r, g, b).

`RGB` (R, G, B) essentially divides (R, G, B) by 255 to return a color triple.

`cmyk`(c, m, y, k) returns the METAPOST color triple equivalent of a cmyk color quadruple.

`gray` (g) returns the color triple (g, g, g), i.e., gray. `gray(0)` is `black` and `gray(1)` is `white`.

All the above commands truncate the color components to the range $[0, 1]$.

`named` (n) returns n if n is a known METAPOST color expression, otherwise `black`. Normally n would be a variable name.

Many of the `grafbase` drawing commands are implemented in `grafbase.mp` as in the following example: The command `safedraw` expands to `colorsafedraw (drawcolor)`, where `colorsafedraw` has a definition almost identical to that of the command `safedraw` in `grafbase.mf`, except for the addition of a color parameter, used with the drawing option `withcolor ....` Such differences account for the ability to set the drawing color with `drawcolor`, the fill color with `fillcolor`, etc. These extra `color...` commands will not be mentioned in the exposition of the basic `grafbase` commands late. Search the file `grafbase.mp` for commands beginning with "`color`" to get the whole story.

Contour to Picture — Clipping and Filling

interior, interiors. clipto, clipsto, clip. picfill, picunfill, picneg.

`interior` c returns the `picture` comprising the *filled* interior of the closed curve (aka *contour*) c, where c is described in *graph* coordinates. *Note:* `interior` is adapted from *The* METAFONT*book*'s `safefill` macro.

`interiors` cc returns the `picture` comprising the *filled* interiors of the *contours* in the `path` array cc, which are described in *graph* coordinates.

The remaining picture operations are derived from the preceding ones. The *contours* are described in *device* coordinates.

`clipto` (v) c clips `picture` v to the interior of *contour* c.

`clipsto` (v) cc clips `picture` v to the interiors of the *contours* in the `path` array cc.

`clip` (v) c returns the `picture` formed by *clipping* `picture` v to the interior of `contour` c. In METAPOST, `clip` is a primative command, so this operation is named `clipped`. Note: in `grafbase.mf`, a side effect of `clip` is that `v` is operated on by `mono`.

`picfill` (v) c changes `picture` v by *filling* the interior of *contour* c. In METAPOST, the color used is `fillcolor`.

`picunfill` (v) c changes `picture` v by *unfilling* (making all pixels zero in) the interior of *contour* c. In METAPOST, this simply fills with the color `background`, which is set equal to `white` in `plain.mp`

`picneg` (v) c returns the `picture` of the reverse video of `picture` v as constrained within the interior of *contour* c. *Note:* An unrestricted reverse video is not possible because that would require replacing all white pixels, out to infinity, by black pixels. In METAPOST, this is a little different. What it does is fill the contour c with `fillcolor` and then draw v on top of that in white. What makes this different is that PostScript makes a distinction between something that is drawn in white and regions that were white because nothing was drawn there. For example, if v is created by filling a large square and unfilling a smaller square inside it, then the entire square is still filled, but part of it is filled with `white`. If you `picneg` (v) c, you just get a large white square inside c.

Rendering Paths — Drawing and Filling

shpath. minpenwd; picpath; picdraw. safedraw, safefill, safeunfill; drawn, filled, unfilled.

All `path`s in this section are in *device* coordinates.

In METAPOST, several of these have variants that can specify a color.

`shpath` (v, q, f) *draws* `path` f in `picture` v using `pen` q. (The 'sh' in its name refers to the initials of 'shade' and 'hatch', where `shpath` is used.)

`minpenwd` is an `internal` variable storing the smallest allowed diameter for a `pen` that's to be used in `picdraw` (see below). `minpenwd` is initially `0.01pt`.

The following series of commands take a `path` argument that is given in *device* coordinates.

picpath d returns a picture containing a (robust) *drawing* of path d.

picdraw (v) d *draws* a path d in picture v. It is designed to be more robust than plain METAFONT's draw macro.

safedraw d *draws* path d in the picture active_plane.

safefill c *fills* the interior of *contour* c in active_plane. *Note:* This is different from the safefill in *The* METAFONT*book*.

safeunfill c *unfills* the interior of *contour* c in active_plane.

The next group of commands take a path argument that is given in *graph* coordinates.

drawn f *draws* the path f (in active_plane), and returns f.

filled c *fills* the *contour* c, and returns c.

unfilled c *unfills* the *contour* c, and returns c. In METAPOST this means filling with white.

- *Note*

In subsequent sections of this document, *drawing* and *filling* write to active_plane.

- *Shading and Hatching*

setdot; onedot, picdot. showbox, bbox; shade; thatchf, thatch, hhatch, vhatch, lhatch, rhatch, xhatch.

Note: Shading and hatching macros *fill* closed paths, but *draw* open paths.

setdot (f, s) returns a picture consisting of the interior of the path f scaled s if f is a closed path, otherwise it returns a drawing of f.

grafbase sets onedot to setdot (dotpath, 0.5pt), shadedot to setdot (dotpath, shadewd), and thepolkadot to setdot (dotpath, polkadotwd).

picdot (v, w, p) superimposes the picture w at the position given by pair p in *device* coordinates on the picture v.

For debugging purposes, grafbase defines a boolean variable showbox which determines whether bounding boxes are shown. Normally, showbox is set to false.

tightbbox (g, ll, ur) sets the pair variables ll and ur to the lower left and upper right corners, respectively, of the *tight* bounding box of the path g. All three arguments are described in *device* coordinates.

tbbox (g, ll, ur) sets the pair variables ll and ur to the lower left and upper right corners, respectively, of the *tight* bounding box containing the array $g[]$ of paths. All three arguments are described in *device* coordinates.

In grafbase.mf, bbox (g, ll, ur) sets the pair variables ll and ur to the lower left and upper right corners, respectively, of the *loose* bounding box defined by the control points of the path g. This is faster than tightbbox and is used when all one needs is an enclosing rectangle (for example, the hatching commands hatch this loose rectangle and then clip to g).

METAPOST already has a bbox command in plain.mp which is incompatible with the above command. Moreover, tightbbox simply calls the METAPOST primatives llcorner and urcorner. If using grafbase.mp, simply use tightbbox where one might use bbox in grafbase.mf.

shade (sp) f shades the interior of path f (described in *graph* coordinates) using dots of a shape and size detrmined by the setdot macro, spaced sp *pixel* units apart.

Successive rows are offset by half of `sp`. The dots are first drawn on a rectangle determined by `ll` and `ur` calculated from `bbox (`f`, ll, ur)`. They are positioned on a pixel grid in this rectangle, and then clipped to f. `shade (sp)` f returns f.

In METAPOST, `shade` merely computes a level of gray from `sp` and `shadewd` and then calls `colorsafefill` with that level of gray.

`thatchf (`v_c`, CT, sp, xa, xb, ya, yb)` hatches the interior of the upright box defined by the X and Y boundaries `xa, xb, ya, yb`, using lines that are horizontal and spaced `sp` units apart, where all five dimensions are measured in the coordinate system determined by the affine transform `CT`. The hatched drawing is added to the `picture` v_c. The line thickness is determined by `hatchpen`.

In METAPOST, an additional color parameter comes between v_c and `CT`.

`thatch (sp,` θ`)` f hatches the interior of `path` f (described in *graph* coordinates) with lines at angle θ, spaced sp apart in *device* coordinates, and adds the drawing to `active_plane`.

`hhatch (sp)` f (horizontal hatching), `vhatch (sp)` f (vertical hatching), `lhatch (sp)` f (left hatching), and `rhatch (sp)` f (right hatching) are special cases of `thatch` where the angle θ is 0, 90, -45 and 45 degrees, respectively. In METAPOST, these hatch in `hatchcolor`.

`xhatch (sp)` f is a cross-hatch that combines left and right hatching.

- *Tiles*

tile, endtile; is_tile; tess.

`tile (atile, unit, width, height, clipon)` ... `endtile` provides an environment in which the tile `atile` can be defined. Within this environment, `active_plane` means the tile, and the unit of length is `unit`. The nominal boundary of the tile is specified by the given `width` and `height`, with the other two sides being at $X = 0$ and $Y = 0$. The tile is clipped to the boundary if the `boolean` expression `clip` is true.

`is_tile (atile)` returns `true` if `atile` is of the correct data type to be a tile, otherwise it returns `false`.

`tess (atile)` c tiles (tesselates) the interior of closed `path` c with an array of copies of the tile `atile`, then returns c. The `path` c is described in *graph* coordinates.

- *Dots and Dashes*

DASHED (dashed), dotted; doplot, gendashed; dashpat.

`DASHED` (d, s) f *draws* a series of dashes along `path` f, with dash length d and dash space s, and returns f. Here d and s are in *device* coordinates, and f is in *graph* coordinates. In METAPOST, `dashed` is a primative, which is why `DASHED` is used instead. In METAFONT, `dashed` is provided as an alias for `DASHED`. `DASHED` does nothing but call `dashpat` and then `gendashed`.

`dotted` (d, s) f *draws* a series of dots along `path` f, with dot size d and dot space s, and returns f. Here d and s are in *device* coordinates, and f is in *graph* coordinates. `dotted` merely calls the more general `doplot` macro.

`doplot (sym, sc, dgap)` f plots the curve f (in graph coordinates) using symbol `sym`, scaled by `sc`, spaced apart by `dgap`. Both `sc` and `dgap` should be in device coordinates. `doplot` initializes `dotpath` to `sym`, sets `dotscale` to `sc`, calls `dashpat (dots) (0, dgap)` and then `gendashed (dots)` f . `sym` should be a path, and it should be closed if a solid (i.e., filled) shape is desired.

`dashpat (pat)` (t) takes a text list of dimensions t and a suffix `pat` and creates three numeric arrays: `pat.start[]`, `pat.rep[]`, and `pat.finish[]`. The main one is `pat.rep` which is used by `gendashed` to draw the repeating pattern of dashes along a curve. It is just the text t copied to a numeric array. The other two are an initial and final part of this pattern. These are used by `gendashed` to draw the start and end of a curve.

`gendashed (pat)` f uses the arrays `pat.start[]`, `pat.rep[]`, and `pat.finish[]`. The odd positions of each array give the length of a dash (in device coordinates), while the even positions give the length of a space (there must be an even number of positions except in `pat.finish`, which ends with a dash). A dash of length 0 is drawn as a dot. The size and shape of the dot are `dotscale` and `dotpath`, which may be reset for special effects. If `pat` is created by `dashpat`, the curve will start with half of the first dash in `pat.rep` and end with half of the same dash, with a whole number of repeated patterns in between. For special effects at the start and end, `pat.start` and `pat.finish` can be defined independently.

- *Points*

bpoint, pointd, plotsymbol.

`bpoint` $(w,\ a)$ returns a `path` that represents a point with diameter w in *device* coordinates and location a in *devive* coordinates.

`pointd` (d, b, t) *draws* a point at each of the *graph* coordinate `pair`s given by the text list t. Each disc has diameter d in *device* coordinates. If the `boolean` value b is `true`, then the disc is filled, otherwise it is white with a boundary. In METAPOST, when the disc is filled, `fillcolor` is used, and when not filled, the boundary is drawn in `drawcolor`.

`plotsymbol` (s, d, t) draws the symbol s with "diameter" d in *device* units at the points given by *graph* coordinate pairs in the text list t. As the symbols are not all round, d is a sort of average diameter. A "symbol" is either one of the predefined paths `Triangle`, etc., listed in Common Global Variables, or a user defined path which should have roughly diameter 1 and be roughly centered at the origin. If the symbol is a closed path, the symbol is filled (in `fillcolor` in METAPOST), otherwise it is simply drawn (in `drawcolor`).

- *Arrows*

hdwdr, hdten, hfilled. headshape, head, headpath. arrowdraw.

`hdwdr` is an internal variable giving the ratio of the width of the arrowhead divided by its length.

`hdten` is an internal variable giving the tension on the curved sides, or *barbs*, of the arrowhead. This controls the curvature of the *barbs*.

`hfilled` is a `boolean` variable. If it's `true`, then arrowheads will be *filled* (in `headcolor`), otherwise they'll be simply drawn (in `headcolor`). If unfilled, and arrowhead is simply two barbs; the ends of the barbs are not connected.

grafbase initialises `hdwdr` to 1, `hdten` to 1, and `hfilled` to false.

`headshape (wr, tens, fil)` sets the global parameters `hdwdr, hdten, hfilled` to `wr, tens, fil` respectively. The initial arrowhead shape is set by `headshape (1, 1, false)`.

`head (front, back, width, tens, filled)` *draws* the arrowhead. The argument `front` is a `pair` giving the position (in *graph* coordinates) of the tip of the arrowhead, `back` is a `pair` giving the position of the base of the arrowhead, `width` is a `numeric` value giving the ratio of the width to the length of the arrowhead, `tens` is a `numeric` value giving the tension of the *barbs*, and `filled` is a `boolean` value which if `true` means that the arrowhead is *filled* but if `false` causes the arrowhead to be simply drawn.

`headpath` (l, θ, b) f *draws* an arrowhead on the path f. The arrowhead has length equal to l *graph* units, it is rotated by θ degrees, and is set back from the last point of f by b *graph* units. The `width`, `tens` and `filled` values of the arrowhead are set by the global variables `hdwdr`, `hdten`, and `hfilled`. `headpath returns the path` f.

`arrowdraw` (l, f) *draws* f and then adds the arrowhead (essentially the same as `headpath` $(l, 0, 0\texttt{pt})$ f, but without returning f).

- *Axes, Axis Tic Marks, and Grid*

axes; xmarks, ymarks; grid.

`axes` (l) *draws* the X axis from *graph* coordinate `xneg` to `xpos`, and the Y axis from *graph* coordinate `yneg` to `ypos`, both with arrowheads of length l *device* units.

`xmarks` (l, t) *draws* tic marks along the X axis, at the X values (in *graph* coordinates) given by the text list t. Each tic extends $l/2$ *device* units above and below the X axis.

`ymarks` (l, t) *draws* tic marks along the Y axis, at the Y values (in *graph* coordinates) given by the text list t. Each tic extends $l/2$ *device* units left and right of the axis.

`grid` (x_s, y_s) draws dots at each grid coordinate at X spacings x_s and Y spacings y_s, in *graph* coordinates.

- *Note*

Unless or until otherwise indicated, subsequent graphics commands are coordinate-independent.

- *Upright Rectangles*

rect.

`rect (ll, ur)` returns the rectangular `path` that has the `pair ll` as its lower left corner, and the `pair ur` as it supper right corner.

- *Path Construction*

mkpath.

`mkpath` (s, c, p) returns the smooth path `mksmooth` (c, p) if the `boolean` value s is `true`, otherwise it returns the polyline `mkpoly` (c, p).

- *Polylines, including Polygons*

mkpoly, polyline.

`mkpoly` (c, p) returns a polyline `path` with vertices given by the array of `pairs` p. The `numeric` variable p contains the number of elements in p`[]`, numbered from 1 to p. If the `boolean` value c is `true`, then the `path` is a closed polygon, otherwise it is an open polyline.

`polyline` (c, t) returns the polyline `path` that has the vertices specified in the list t. If the `boolean` value c is `true`, then the `path` is a closed polygon, otherwise it is an open polyline.

• *Smooth Curves*

mksmooth, curve mkcontrolledfcn, functioncurve, openqbs, closedqbs. mkopencbs, opencbs, mkclosedcbs, closedcbs.

`mksmooth` (c, p) returns a smooth curve passing through the points given by the array of `pairs` p. If the `boolean` value c is `true`, then the curve is smoothly closed, otherwise it is open. The curve is defined to have tension (in the METAFONT sense of tension) between each pair of points equal to the `curvetension`, a global numeric parameter.

`curve` (T, c, t) returns a smooth, curved `path` that passes through the `pairs` given in the text list t. If c is `true`, then the `path` is closed, otherwise it is open. The curve is defined to have tension T.

`mkcontrolledfcn` (p) returns a smooth curve through the points given by the array of pairs p. The curve is "controlled" by a numeric parameter `functiontension`. If `functiontension` is greater than or equal to 1, *and* the x coordinates of the points in p are increasing, then the x coordinates of the curve are guaranteed to increase. Therefore, the curve will be the graph of a function. The parameter `functiontension` is not strictly a tension in the METAFONT sense, but functions similarly to influence the location of control points.

`functioncurve` (T, t) sets `functiontension` to T, converts the text list of pairs t to an array p, and then calls `mkcontrolledfcn` (p).

`openqbs` (t) returns an open quadratic B-spline `path` governed by the `pair` list in the text t.

`closedqbs` (t) returns a closed quadratic B-spline `path` governed by the `pair` list in the text t.

`mkopencbs` (b) returns an open cubic B-spline `path` governed by the `pair` array b.

`opencbs` (t) returns an open cubic B-spline `path` governed by the `pair` list in the text t.

`mkclosedcbs` (b) returns a closed cubic B-spline `path` governed by the `pair` array b.

`closedcbs` (t) returns a closed cubic B-spline `path` governed by the `pair` list in the text t.

• *Path Closure*

lclosed; sclosed; bclosed; uclosed; ztob, cbclosed.

If f is a closed `path`, then `lclosed` f returns f; otherwise it returns the closure of f by a straight line.

`sclosed` f returns the result of closing `path` f in the manner of `mksmooth`. The curve returned by `sclosed` f may differ in shape from f, due to METAFONT optimization of control points over the whole of a curve.

`bclosed` f is like `lclosed` f, except that it closes using a METAFONT Bézier curve. *Note:* The curve returned by `bclosed` f may differ in shape from f, due to the `tension` exerted by the Bézier.

`uclosed` f closes f in a smooth manner, but unlike the previous two operations, does not change the shape of f.

`ztob` (z, b) converts the four Bézier segment key points in the `pair` array z into four cubic B-spline control points in the `pair` array b.

`cbclosed` f returns the `path` formed by closing the given `path` f by a cubic B-spline. (This macro uses `ztob`.)

- *Circles and Ellipses*

ellipse, circle.

`ellipse` (c, r_x, r_y, θ) returns an ellipse with center at the *graph* coordinate `pair` c, 'X' radius r_x, 'Y' radius r_y, and with its 'X' radius angled at θ degrees from the *graph* coordinate system's X axis.

`circle` (c, r) returns a circle with center at the *graph* coordinate `pair` c and radius r.

- *Circular Arcs*

arc; arccenter; arcpps, arcplr, arccps, arcppp.

`arc` (c, p_1, θ) returns a `path` describing a circular arc for a circle with center at the point c, starting point at p_1, and sweep angle θ in degrees.

`arccenter` (p_1, p_2, θ) returns the `pair` defining the center of the circle that has the points p_1 and p_2 on its circumference separated by an angle of θ degrees.

`arcpps` (p_1, p_2, θ) returns a `path` describing the arc that has starting point p_1, finishing point p_2, and sweep angle θ degrees.

`arcplr` $(c, \theta_1, \theta_2, r)$ returns a `path` describing the arc that has center point c, starting direction θ_1, finishing direction θ_2, and radius r.

`arccps` (c, p_1, θ) returns a `path` describing the arc that has center point c, starting point p_1, and sweep angle θ degrees.

`arcppp` (p_1, p_2, p_3) returns a `path` describing the arc that has the given starting point p_1, finishing point p_3, and that passes through the point p_2.

- *Polar Coordinates*

polar.

`polar` (p) returns the rectangular (cartesian) coordinate `pair` (x, y) equivalent to the polar coordinate `pair` $p = (r, \theta)$.

- *Turtle*

turtle.

`turtle` $(p_0, v_1, v_2, \ldots)$ returns a `path` that starts at the point p_0, then moves by v_1, then by v_2, and so on. (How old do you have to be to remember "turtle graphics"?)

- *Sectors*

sector.

`sector` $(c, r, \theta_1, \theta_2)$ returns a `path` describing a circular sector boundary with center point c, radius r, starting angle θ_1, and finishing angle θ_2.

- *Utility Functions*

  id.

  `id` (x) returns x. This is the identity function for the next topic.

- *Functions and paths*

  mkfcn, tfcn, parafcn; xfcn, function; rfcn, plrfcn.

  Note: Functions may be user-defined.

  `mkfcn` (s, a, b, d, pf) returns a `path` passing through the X and Y coordinate `pairs` traced out by $pf(v)$ for each value v from a to b stepping by d. Here pf may be any METAFONT `pair` function of one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline. The step d will be adjusted if necessary so that a whole number of steps takes place, and the curve ends with v precisely equal to b.

  `tfcn` is a synonym for `mkfcn`.

  `parafcn` (s, a, b, d, pf_t) returns a `path` passing through the X and Y coordinate `pairs` traced out by the text pf_t, which is a *literal expression* in t, for each value t from a to b stepping by d. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline. `parafcn` uses the text pf_t to define a function pf and then calls `mkfcn`.

  `xfcn` (s, a, b, d, f) returns a `path` passing through the X and Y coordinate `pairs` $(x, f(x))$ for each value x from a to b stepping by d. Here f may be any METAFONT `numeric` function of one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline. This `xfcn` merely calls `mkfcn` with pf defined by $pf(x) = (x, f(x))$.

  `function` (s, a, b, d, f_x) returns a `path` passing through the X and Y coordinate `pairs` (x, f_x), where the text f_x is a *literal expression* in x, for each value x from a to b stepping by d. Here f may be any METAFONT `numeric` function of one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline. `function` uses the text f_x to define pf and then calls `mkfcn`.

  `rfcn` (s, a, b, d, f) returns a `path` passing through the X and Y coordinate `pairs` $f(t) *$ (`dir` t) for each value t from a to b stepping by d. (*Note:* `dir` t = `(cosd (`t`), sind (`t`))`.) Here f may be any METAFONT `numeric` function in one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline. `rfcn` constructs a pair-valued function as described and then calls `mkfcn`.

  `plrfcn` (s, a, b, d, f_t) returns a `path` passing through the X and Y coordinate `pairs` $f_t *$ (`dir` t), where the text f_t is a *literal expression* in t, for each value t from a to b stepping by d. (*Note:* `dir` t = `(cosd (`t`), sind (`t`))`.) Here f may be any METAFONT `numeric` function in one `numeric` argument. If the `boolean` s is `true`, then the `path` is a smooth curve, otherwise it is a polyline. `plrfcn` constructs a pair-valued function as described and then calls `mkfcn`.

- *Text labels (*METAPOST *only)*

  gblabel

  `gblabel` (l, r, b, t, a) (s, p) places the label s at the point p, positioned according to the information in (l, r, b, t, a). If L, R, are the x-coordinates of the left and right end of the bounding box of s, and B, and T are the y-coordinates of the bottom and top, then the text is shifted so the point $(lL + rR, bB + tT)$ is at the origin (normally the leftmost point of the baseline is at the origin), then rotated by angle a, then shifted to the point p. For example, to place the lower left corner of s at p, use $(1, 0, 1, 0, a)$. To place the center point of the baseline of s at p, use $(1/2, 1/2, 0, 0, a)$. Note: One can think of l, r, b, t as selecting a new reference point, after which the text is rotated about it and then moved so the reference point is at p.

The text s should be either of type string or picture. The most common example would be a `btex ... etex` expression (which is syntactically a picture expression). If s is a string, it is converted to a picture with s `infont defaultfont scaled defaultscale`.

- *Overlays*

Global Parameters: *totalpicture, totalnull, currentnull.*
Macros: clearit, keepit, addto_currentpicture, mergeit, shipit, showit_, show_.

Essentially these redefine the middle-level plain METAFONT macros for the formation of the current font character, so that typing `keepit;` will preserve work-to-date from being destroyed by subsequent erasures on subsequent parts of the same character.

The parameters are for bookkeeping and should not be touched by users. Those macros intended for public use are `clearit, keepit, mergeit, and shipit`. Of these, only `keepit` is new; the others behave in a similar way to their descriptions in *The* METAFONT*book*.

For a more detailed description of these overlay macros (devised by Bruce Leban), see *The* METAFONT*book* page 295.