

# A mkdrawf tutorial


This is a fairly gentle introduction to `mkdrawf`, a program for creating *drawfiles*. If you don't know what a drawfile is, don't worry: you will soon. If you don't know what a program is, perhaps you should read some other tutorials first. In fact, even if that's the case you can probably get a fair way into this tutorial, since most of it consists of instructions saying "Put this into a file and do that with it, and see what happens".


When you've worked through this tutorial, you should find most of the manual pretty easy going. Actually most of the manual is easy going anyway, but it suffers a bit from being intended as a reference as well as a tutorial; this document has no such ambitions, and is unashamedly incomplete; it even contains a few (minor) lies. If it disagrees with the manual, you know which to trust.

## Running *mkdrawf*

`mkdrawf` manufactures drawfiles out of ordinary text files. There are two ways to use it. *Firstly*, you can run it from the command line: entering a command like

```
mkdrawf textfile drawfile
```

will process the file `textfile` and produce an output file `drawfile`. *Secondly*, you can use the Wimp application `!Drawf`, which allows you to drag a text file to its icon, whereupon it will run that file through `mkdrawf` and allow you to drag the resulting drawfile somewhere to save it. The icon for `!Drawf` looks like this: .

(As you might guess from the  look of the icon, `!Drawf` will also decode drawfiles for you, producing output suitable for handing to `mkdrawf` again. There is a separate manual for `!Drawf` which will tell you all about this, and more besides.)

From now on, I shall assume that you can do one of these two things. When I say "run `mkdrawf` on this file", this means: either choose an output filename and use the command-line, or else drag the file to the `!Drawf` icon and put the resulting output somewhere.

## Running *!Draw*

The easiest way to see the results of using `mkdrawf` is to view the files it produces using `!Draw`. This program comes with every Archimedes or Risc PC, so you should definitely have a copy; you can probably find it by clicking on an icon labelled `Apps` at the left-hand side of your icon bar.

If you haven't used `!Draw` before, the first thing you should do is to run it and play around with it for a while. This should give you an idea of the sort of thing it can do.

A *drawfile* is a file which `!Draw` can understand. A drawfile consists basically of a number of *objects* (lines, curves, bits of text, that sort of thing) strung together. In some cases (have you used the *Group* option on the menu?) an object can contain a number of other objects. As this tutorial proceeds, you will learn rather more about drawfiles than you actually want to know.

Every time you run `mkdrawf` you should have a look at the output by either double-clicking on the file it produces (*after* dragging it to a directory display, if you are using `!Drawf`) or dragging the file to `!Draw`'s icon on the icon bar. (Not quite every time; if something goes wrong and you get

lots of error messages, the resulting drawfile may be full of rubbish, or at any rate not full of what you wanted.)

### *A very simple example*

Put the following into a file using your favourite text editor, run it through `mkdrawf`, and look at the result using `!Draw`:

```
# This is a comment; mkdrawf will ignore it.
Path {
  Move 100 100
  Line 300 200
}
```

You can probably guess what the result of this will be before you try it. Your guess will almost certainly be correct. This simple example actually demonstrates quite a few things about `mkdrawf`, though... The first line is self-explanatory. The remaining lines describe a single *object*. An object is introduced by saying what sort of object it is; in this case, it is a *path object*. (The typical drawfile consists mostly of path objects.) Further details about the object are given within those braces `{}`; in this case, the path consists of a single line segment from (100,100) to (300,200). Coordinates are always given, as in this example, as pairs of numbers. The unit, by the way, is the *point*; the ambiguity here is unfortunate but seldom causes trouble in practice. A point is 1/72 of an inch; the spacing between vertical lines here `|+++++|` is about 10 points.

### *A slightly less simple example*

As I already said, a drawfile typically contains several objects, one after another. This structure is represented in the obvious way in input to `mkdrawf`; namely, by putting one object description after another. Here's an example, which (as usual) you should try.

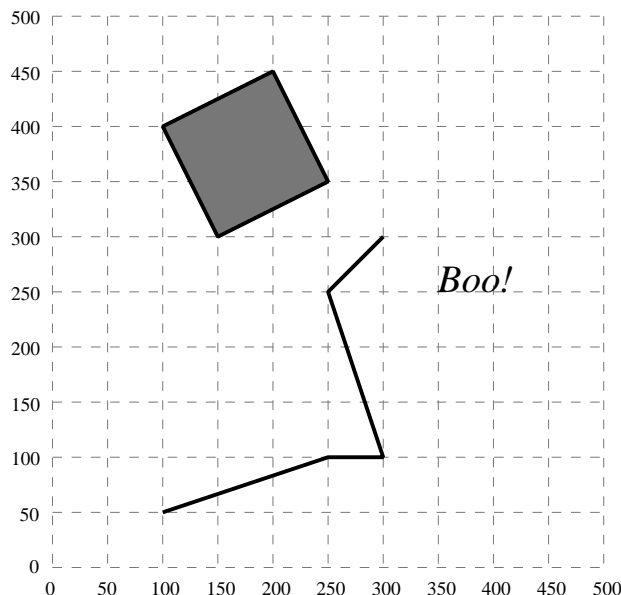
```
Path {
  Move 100 100
  Line 150 100
}
Path {
  FillColour r220g0b0
  Move 200 200
  Line 300 200
  Line 300 300
  Close
}
Text {
  Style Font "Trinity.Medium"
  StartAt 200 300
  Size 10 10
  Text "Hello!"
}
```

This produces a drawfile containing three objects: two path objects and a text object. You can probably make a pretty good guess as to what this will look like, if I tell you that:

- colours are given as RGB values, each component going from 0 (dark) to 255 (bright);
- Size 10 10 means “I want a 10-point font”;
- *Trinity* is the typeface in which this tutorial is printed.

## An easy exercise

Write a `mkdrawf` file which, when fed into `mkdrawf` and `!Draw`, produces the following picture. You are not expected, of course, to produce the grid lines and the numbers labelling them (though you might be able to do something approximating to them), and you should not worry about the thicknesses of the lines. The grey colour inside the square is `r119g119b119`, I think; and the text is 20-point *Trinity.Medium.Italic*. When you've done this, check it (of course you won't be able to check the scale) by running it through `mkdrawf` and `!Draw`.



## More about path objects

You may have guessed from the previous picture (or you may have noticed while using `!Draw`) that all sorts of things we haven't discussed yet are possible with path objects; for instance, the picture includes dashed lines, lines in colours other than black, and lines of different thicknesses. And there's plenty more. Anyway, here is a `mkdrawf` file demonstrating some more features of path objects:

```
Path {
  Width 2                                # all dimensions, including this, are in points
  OutlineColour r0g0b255                 # blue lines
  Style { EndCap Triangular }            # !Draw will show you what this means
  Move 100 100
  Line 200 100
  Curve 300 100 200 200 200 300          # a Bezier curve, ending at (200,300)
  Line 200 400
  Move 200 100                            # a path can be made up of many subpaths
  Line 200 300
}
```

What's new here? The `Width` keyword allows you to say how thick you want a line to be. The width is, like all dimensions, given in points (1/72"). It's a "diameter" rather than a "radius". The `Style` keyword should be followed by some stuff in braces; there are several other things you can set in there. The `OutlineColour` keyword (so called in contrast to `FillColour`) determines the colour of lines and curves, as distinct from the filled-in space inside them. By the way, the usual 16-colour desktop palette doesn't include a colour accurately matching `r0g0b255`, but `!Draw` will

happily display the best approximation it can, and won't throw away its information about *exactly* what colour you really wanted.

The line beginning Curve is more interesting. As well as straight lines, a path object can contain *Bezier curves*. A Bezier curve is described by giving its starting and ending points — in this case (200,100) and (200,300) — and two *control points* — in this case (300,100) and (200,200). The curve starts out from its starting point, heading towards its first control point. It curves around until eventually it reaches its ending point, from the direction of its second control point. Subject to these restrictions, the exact shape of the curve depends on how far away the control points are from their matching endpoints. Got that?

I think the best way to get some intuition for Bezier curves is: Go into !Draw, select the “open polygon” tool (the one at the very top of the toolbox), click somewhere, click somewhere else, and then click with the *Adjust* (right-hand) mouse button. At this point you should see a grey line with a square at each end. Now, click with *Adjust* on the square at whichever end of the line you clicked on second; the whole line should go red. Click with the *Menu* button; the menu you get should include an option “Change to curve”. Do this. Now you should see, as well as the two endpoints of the line, two other points in the middle of it. These are the control points of the Bezier curve you have just produced. Drag them around with the *Adjust* button and watch the shape of the curve change. (If you accidentally click somewhere you didn't mean to and the coloured squares disappear, then select the line again using the “arrow” tool and hit control-E.)

No, really. You should actually do this, not just read about it. Otherwise you'll never really understand Bezier curves, and they're important.

## *Doing things again and again and again*

There are several other sorts of object, but I think it's time to look at some other features of `mkdrawf` before getting even further bogged down in the details of what they are and how to specify them. Here's one of the problems that first got me working on writing this program. I wanted to be able to draw graphs of functions (sine, cosine, that sort of thing) and make them into drawfiles, so that I could add annotations, print them out and so on. Now, obviously we *could* do that by writing an enormously long `mkdrawf` “script” looking something like

```
Path {
  Move 100 400
  Line 100.1 400.097
  Line 100.2 400.193
  blah blah blah
  Line 500 400
}
```

with hundreds or thousands of lines, but this would be horrible. But try feeding the following program into `mkdrawf`... (This introduces quite a lot of new ideas, so don't worry if you can't see what it does yet. Have a look at the drawfile it produces anyway.)

```
Set $2Pi 6.28318530717959
Set $Factor Over 400 $2Pi
Path {
  Width 1
  Move 100 400
  For $x0 0 200 {
    Set $x Times $x0 Over $2Pi 200
    Set $y Sin $x
    Set $t Plus 100 Times $x $Factor
    Set $u Plus 400 Times $y $Factor
```

```

    Line $t $u
  }
  Line 500 400
}

```

OK. Let's take this slowly, since there are lots of tricky things in it. Anything starting with a dollar sign is a *variable*. You can set the value of a variable to be anything at all (including colours like `r123g234b11`, strings like `"Trinity.Medium"`, and other even stranger things), but most variables are used to contain numbers. So, the first line of the program says "Until further notice, any time you see `$2Pi` you should replace it with the number 6.2831853071959."

The second line introduces arithmetic. I'm afraid the syntax for this is horrible; its motivating principle is that considering the amount of effort I've put into the program as a whole, you're lucky to get arithmetic at all and you have no right to complain if it doesn't look very nice... More seriously, the principle is that you have to say *what to do* before saying *what to do it with*. So "`2+2`" violates this rule, because reading from left to right you see the first "`2`" before you know that you're going to have to add it to something; to make `mkdrawf`'s life easier, you have to say `Plus` first. Sorry. Anyway, the second line divides 400 by `$2Pi`'s value, and puts the result into a variable called `$Factor`.

The next few lines are familiar: they start a path object, indicate the width of the line to be drawn, and move to the starting position.

What happens next is altogether new. The `For` line means, approximately, "Do everything inside the `{ }` once with `$x0` having the value 0, then again with it having the value 1, then ... and finally with it having the value 199." In other words, we add 1 to `$x0`'s value every time, and we give up when it reaches 200. So, that's the significance of the magic numbers 0 and 200 on that line: start at 0, give up at 200.

If you have a look inside those braces, you'll see that the net result is a lot of calculation and 200 Lines. I'll let you work out exactly what the calculation is doing — it's not very difficult, especially with the resulting drawfile in a `!Draw` window on your screen. (You *did* run it through `mkdrawf`, didn't you?) Actually, it would be unkind to leave you with no help about the nasty arithmetic things. The first line inside the `{ }`, for instance, sets `$x` to what in any civilised computer language would be called `$x0*($2Pi/200)`.

Finally, we draw a line to the right ending point (we should be very near to it at the end of the `For` loop anyway), and that's the end.

(Incidentally, you may be wondering: Why `For` rather than `Repeat` or `ManyTimes` or something? Answer: it's sort-of traditional that this sort of construct is called a "for loop". In the first computer languages to use the word "for" for this sort of thing, the syntax looked more like

```

  For x = 1 to 100 do blah blah blah end.

```

which you could read as "Do *blah blah blah* for `x=1`, then for `x=2`, and so on". This at least makes some sort of sense. This sort of syntax lives on in BASIC and Pascal.)

Got all that? If not, don't worry. You can always come back to it.

## Another exercise

Work out what the following does. It is intended to be put in the same `mkdrawf` input file as the sine-curve we just drew, but don't try it until you think you know what will happen.

```

Path {
  Width 0          # this means: as thin as possible
  Move 100 400
  Line 500 400
  Move 100 Minus 400 $Factor
}

```

```

Line 100 Plus 400 $Factor
# every pi/4 units:
For $n 0 9 {
  Set $x Times $n Over $2Pi 8
  Set $t Plus 100 Times $x $Factor
  Move $t 390
  Line $t 410
}
}

```

### *And another exercise*

You should now be able to do this: Produce some squared paper, with lines as thin as possible. The lines should be spaced at intervals of 10 points, and occupy the rectangle whose bottom-left and top-right corners are (100,100) and (500,500).

You will need two loops. I recommend, as a matter of style, having one object containing the horizontal lines and one containing the vertical lines. Be very careful about the numbers in your Fors; remember that the second number is an *exclusive* rather than an *inclusive* limit. (Actually you don't *need* two loops. As another exercise, work out a way of doing it with one. Don't bother to implement this.)

### *Macros*

Consider the following situation. You want to draw a diagram in which a number of points are marked with little crosses, thus: +. If there are, say, 100 points then this could mean an awful lot of typing. You can save a lot of effort by using a *macro*; what this means is best illustrated by an example.

```

Define Point {
  Path {
    Width 0.3
    Move Minus %x 2 %y RLine 4 0
    Move %x Minus %y 2 RLine 0 4
  }
}
Point { %x 100 %y 200 }
Point { %x 200 %y 300 }
Point { %x 123 %y 321 }

```

There are three new features here. The first, trivial, one is the use of the RLine keyword, which is just a labour-saving device; the same result could have been produced by

```

Move Minus %x 2 %y Line Plus %x 2 %y
Move %x Minus %y 2 Line %x Plus %y 2.

```

There are also RMove and RCurve keywords.

The second feature is the use of a macro. The first 7 lines say: Whenever Point appears, replace it with the stuff in lines 2–6, making certain changes. The “certain changes” are the third feature: the first time the macro is “invoked” (on line 8), “%x” will be replaced everywhere by “100” and “%y” by “200”. You can probably guess what will happen the other two times. %x and %y are called “parameters” of the macro. You can find out much more about all this by reading the manual.

I've never worked out why the word “macro” is used for this sort of thing.

## Ellipses and rectangles

The “ellipse” and “rectangle” tools of !Draw don’t actually correspond to special kinds of object. An ellipse or rectangle is just a path which happens to be the right shape. One consequence of this is that ellipses aren’t *really* ellipses, but close approximations by Bezier curves. This is a pain, since ellipses (especially circles) are often what you want, and finding a sequence of Bezier curves that approximates a circle well is not entirely trivial.

Fortunately I’ve already done the work for you. In the manual you will find, as part of one of the example programs, a macro which does just that, together with a couple of examples of its use. Incidentally, reading through it and checking you understand what’s going on (apart from the choice of the magic value for %t, whose justification is half a page of mathematical scribbling) might be a good idea.

## Conditionals

You’re preparing a graph to show your boss, illustrating something terribly important. The graph consists of lots of points plotted with little crosses, and you want some of the points — the ones showing unusual results — plotted in red. To be more precise, let’s suppose that a y-coordinate of 300 or more indicates something wrong, and you want those points plotted in red. You *could* define a RedPoint macro and a BlackPoint macro, and in fact if the mkdwarf file in question is being produced by a computer program that’s probably the way to do it. But if you’re doing it by hand, you might find the extra typing (Red and Black) annoying, and you might not be entirely confident of your ability to make no mistakes. So, instead...

```
Define Point {
  Path {
    Width 0.3
    OutlineColour
      IfLess y 300 r0g0b0
      Else          r255g0b0
    EndIf
    Move Minus %x 2 %y RLine 4 0
    Move %x Minus %y 2 RLine 0 4
  }
}
Point { %x 100 %y 200 }      # this will be in black
Point { %x 200 %y 300 }      # but this will be in red
Point { %x 123 %y 321 }      # and so will this
```

You can have more complicated conditions than this, of course, provided you can massage them into the form “so-and-so is less than such-and-such”, or “so-and-so is equal to such-and-such”, for which you use IfEqual. You should be warned that arithmetic involving anything other than integers is likely to be imprecise, so IfEqual may not behave the way you expect it to. For instance, if you do

```
Define $Pi 3.141592653589
IfEqual Sin $Pi 0
  blah
EndIf
```

you should not rely on *blah* being done. There is another kind of If, called IfExists; see the “arcs of circles” thing in the manual for an example of its use.

### *And yet another exercise*

Write a `mkdrawf` program that produces 200 random points spread uniformly over the square whose bottom left and top right corners are at (100,100) and (200,200), and plots each one with a cross as discussed above, coloured black if it's inside the circle that touches all four sides of the square and red if it's outside it. (The circle in question has centre (150,150) and radius 50. You may colour points on its circumference either red or black; I don't mind.)

### *That's all, folks...*

At this point you have seen most of the tricky things about `mkdrawf`, and you should now read the manual. If you've done all the exercises, this should be a piece of cake.