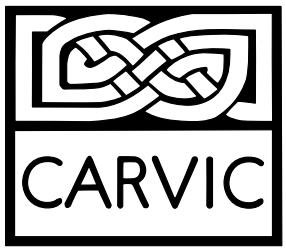


DrawAid 3

Reference Guide

CARVIC Manufacturing
FORRES
Scotland

DrawAid
Reference Guide

Third Edition 14th March 1996
Author: William R. Graham

This application makes use of the Acorn Drawfile module, a copy of which is included inside the application directory. This module is distributed with Acorn's permission, but remains their copyright, which is hereby acknowledged.

Disclaimer While CARVIC Manufacturing has taken care to ensure that the information contained in this document and the DrawAid product is reasonably correct, and since by its nature the software will interact closely with the user's own programs, CARVIC Manufacturing accepts no liability for any consequential loss or damage, however caused. The user must ensure to his own satisfaction that the results of the joint programs are correct.

V3.01 14th March 1996

Note: See ReadMe File for any changes to these conditipns
V3.02 4th January 2003

Preface

Anyone who has used a variety of computers, as I have over many years in industry, will know just how outstanding is the Acorn RISC OS series of machines.

It is true that much of the independent software for the computers is also first class, as is evidenced by the Impression Publisher document processor from Computer Concepts, on which this Guide was prepared.

It is also true that in the past many engineers, scientists, mathematicians and other professionals were adept at writing their own specialist programs as working tools, often in BBC BASIC. However, although the advent of Wimp based, multi-tasking software takes user-friendliness into a new realm it tends to suggest that unless a user's own programs make full use of the Wimp, they cannot be worthwhile. For many one-off industrial problems this leads to a situation analogous to trying to fit a jet engine to a garden strimmer. We should not forget that computers are tools of great scope and flexibility, and that lots of "straight", specialist, little BASIC programs can produce much more added value to a professional than a Wimp solution, which doesn't quite do all that is required, is difficult to modify, and took months of effort.

One need which is central to technical professionals is the preparation of specialist charts, diagrams, and drawings related to their own discipline, and constructed from their own data or calculations. While RISC OS spreadsheets and art packages may provide most of the answers as Draw or sprite files, these often have to be tailored or fudged using the Acorn Draw and Paint applications. Some people manage by producing a drawing from their own program on the screen, and then using Paint to "grab" a sprite for import into their DTP document.

Version 1 of DrawAid was written in 1991 (followed by Version 2 in 1993) to allow professional engineers and others to prepare Draw files directly from their own BASIC programs without knowledge of either the Wimp system or of Draw file structure, and with the minimum of fuss.

Version 3 of the software is now fully RISC OS compliant, making it easier than ever to produce simple programs which generate Draw files to the user's exact needs with the very minimum of work. Further enhancements include messaging and data input, symbols, continuous splines, font and sprite rotation, and better control of line styles.

Because DrawAid is object oriented, and uses everyday units (mm) I have personally found it to be invaluable in my own work, and I am sure that my colleagues and all Acorn RISC OS users will find it to be a useful tool in their professional endeavours.

Forres ; 14th March 1996 *William. R. Graham*

Contents

Introduction

The Acorn RISC OS series of computers are supplied with *Draw*, an excellent, general purpose, object based drawing program. *Draw* allows the mouse to be used interactively to generate high quality line drawings. These drawings can then easily be incorporated into professional documents by use of a Desk Top Publishing package. However, users of *Draw* may have found it difficult at times to obtain accurate drawings, particularly where curves are used, or repetitive positioning is required. For instance, the drawing of items like gear wheels or instrument dials is extremely difficult.

DrawAid is an object-oriented, application environment which is designed to overcome these limitations and allows BASIC programmers to generate complex and accurate *Draw* files directly from their own programs. The facility to produce multiple file output from one BASIC program is useful for producing a series of parametric drawings, with indexing of one or more variables. In addition to providing access to the RISC OS fonts, a special rotating font is supplied which is specially designed for output to pen plotters. Facilities are also provided for the location, and manipulation of sprites, and many of the objects produced by *DrawAid* can be defined by coma separated value or CSV files.

Some users may have only a limited knowledge of BASIC, and *DrawAid* is designed to make it as easy as possible for such users to generate *Draw* files. More advanced users will readily be able to produce their own general purpose programs for such applications as graph plotting, pattern design, the drawing of mathematical functions, and the preparation of drawings of families of electrical and mechanical components. They will be able to produce their own specialist procedure Libraries.

DrawAid operates from the desktop, and consists of two components. The first part is the user's BASIC program which, together with the *DrawAid* BASIC Library, can run as an independent task making maximum use of memory and speed. Alternatively it can run as a fully multi-tasking program in cooperation with the second component, which is the *DrawAid* RISC OS interface.

This Reference Guide describes the facilities provided, and demonstrates how to use *DrawAid* by means of examples.

1: Using DrawAid

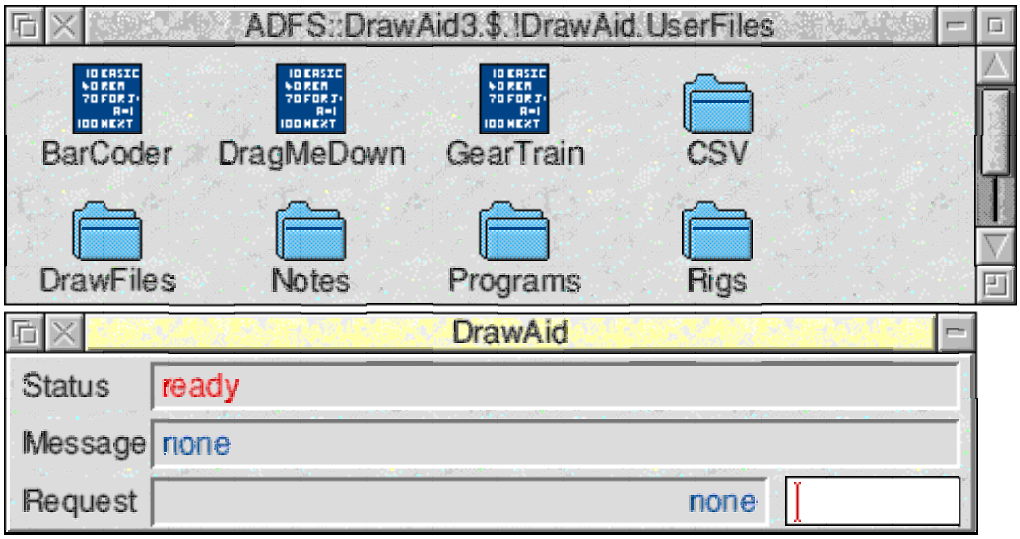
The disc supplied is not write protected, and should be copied onto a working disc and then kept in a secure location. Note that each copy contains a coded individual licence number identifying the licensee. Refer to the title page for copying conditions.

Those with a hard disc should copy *DrawAid* into any suitable directory.

Programs may be prepared using any BASIC editor, perhaps preferably Acorn's *Edit* which is supplied with RISC OS, or commercial editors such as *DeskEdit* or *StrongEd*.

The *System* directory needs to be available to *DrawAid*, and should have been "seen" by the operating system prior to running *DrawAid*.

Double clicking with the mouse select button on the *DrawAid* icon will install the application on the icon bar. In addition it will set up various facilities required by *DrawAid*. Clicking with select on the installed icon will open a long dialogue box or monitoring window, and a Directory viewer called *UserFiles*.



Initially this Directory viewer shows three demonstration BASIC files. and seven directories called *CSV*, *DrawFiles*, *Notes*, *Programs*, *Rigs*, *SpriteFile* and *Tools*.

On your own working disc the directory *Rigs*, and the three BASIC files can later be deleted. The *Programs* directory contains the supplied examples, and you may replace them with your own programs although user programs can be stored anywhere on the system. Likewise the directories *Notes* and *Tools* may be relocated wherever the user wishes. However, *CSV* and *SpriteFile* are private directories for use by *DrawAid* and should not be deleted or moved, unless they are redundant to the user's needs.

DrawFiles should neither be moved nor deleted. These directories should be "cleaned" if large files are to be saved on a working floppy disc.

SpriteFile is where the sprites to be used by the BASIC programs are stored, and *CSV* is similarly where the user's CSV files should be placed. The *DrawFiles* directory is where the resulting *Draw* files will be placed by *DrawAid*. The *Rigs* directory is used by one of the example programs included.

Inside the *Programs* directory will be found a BASIC program called *AidBlank.* This is provided as a starting point for the user's own BASIC programs and should be dragged into *Edit*, or whichever BASIC editor is to be used, when a new BASIC program is to be prepared. See section **3 : Program Format** on how to prepare programs.

When the *DrawAid* icon is installed a *Library* directory, containing all of the *DrawAid* procedures, is made available and is subsequently loaded by the user's BASIC program. The user need not be concerned about this *Library* directory, unless he or she wishes to add their own procedure library, and its contents should not be modified.

Below the *UserFiles* directory viewer is a long dialogue box which is used by *DrawAid* to provide progress messages from both the application and the user's program. Data can also be entered here on request from the user's program. BASIC programs can be "launched" by dragging them either to this monitoring dialogue box, or to the icon bar icon. A further window, used to display the generated *Draw* files, can also be used to start programs. Programs launched in this manner will be treated as sub-tasks of *DrawAid*, and will be multi-tasking. *DrawAid* will monitor their progress, and display sequentially any number of *Draw* files generated.

If a BASIC program is double-clicked, rather than launched as described, it will run as a single task, and only the last drawing it generates will be displayed. If *DrawAid* has been Quit from the icon bar then BASIC files will still run, provided *DrawAid* has been run once to set up its environment. In this case no display will be provided, but the *DrawFiles* directory will open to show the files produced. This method will be found to be faster for "production" programs, and useful for programs requiring maximum memory. The *TaskManager* should be used to set the *Next Task* bar to a suitable value.

In addition to the above functions, any *Draw* file, produced by any other program, dragged to the icon bar *DrawAid* icon, or on to either of its two windows, will be displayed.

The three BASIC files are provided as demonstrations. *DragMeDown* should be run first by dragging it into the dialogue box. *Barcoder* requests a number to be entered into the data icon. Click with select on this icon to acquire the caret, and enter a number up to thirteen digits. *GearTrain* will prepare a drawing of two engaging gear wheels. Two numbers are required, and should be entered into the icon as requested. Typical values are 13 teeth and 28 teeth.

2 : Demonstration Program

If the *DragMeDown* BASIC file is dragged from the *UserFiles* directory to the dialogue box it will provide a demonstration of *DrawAid* in action.

A simplified listing of this program with the messages, comments and timing routine removed is shown below. This is the required essence, but inspect the full program in your editor.

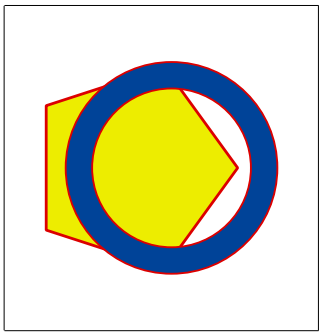
```
REM  >DragMeDown
LIBRARY "<DrawAid_Lib$Dir>.Procedures"
PROC_report_errors
object=1:PROC_DrawAid("Drawing1")
object=2:PROC_DrawAid("Drawing2")
object=3:PROC_DrawAid("Drawing3")
PROC_finish
END

DEFPROC_Objects
xc=50:yc=40:REM set object centre coordinates
IF object=1 THEN
  PROC_circle(width2,black,red,xc,yc,30)
  PROC_outline_text("Trinity.Bold",grey2,red,8,20,22,24,"Welcome",10)
  PROC_outline_text("Trinity.Bold",black,grey2,7,21,22,24,"Welcome",10)
ENDIF
IF object=2 THEN PROC_polygon(width3,red,yellow,xc-5,yc,40,5,0)
IF object=2 THEN PROC_ring(width2,red,dark_blue,xc+10,yc,30,40)
IF object=3 THEN PROC_userproc
ENDPROC

DEFPROC_userproc
REM Draw shaded ellipse with Vector text at an angle
LOCAL xo,yo,majoraxis,minoraxis,angle
xo=xc:yo=yc:majoraxis=70:minoraxis=35:angle=20
PROC_new_group
FOR colour=0 TO 7
   PROC_ellipse(width4,none,colour,xc,yc,majoraxis,minoraxis,angle)
   majoraxis-=2.5:minoraxis-=2.5
   NEXT colour
PROC_end_group
PROC_vector_text(red,xc-38,yc-26,20,bold,oblique,angle,"DrawAid")
PROC_vector_text(yellow,xc-38,yc-26,20,light,oblique,angle,"DrawAid")
ENDPROC
```

When the *DragMeDown* demonstration program is run it will draw attention in the dialogue box, then generate three independent *Draw* files called *Drawing1* to *Drawing3*, and place them in the *DrawFiles* directory. A running progress report is displayed in the dialogue box. The resulting *Draw* files may be inspected again by dragging them into the display window.

Drawing1



Drawing2

Drawing3

If you happen to start the *DragMeDown* program by single clicking on its file, it may appear that your machine has stopped, but the program is single tasking and will conclude by opening the *DrawFiles* directory. You can stop single tasking programs by pressing the <escape> key. Multi-tasking programs can be stopped by opening the *TaskManager*, selecting *DrawAidTask* with the Menu button, and Quitting. Note that quitting *DrawAid* from the icon bar does not stop any subtask which is running.

Note that, for ease of use, the default units used in *DrawAid* are **mm** and **degrees**. All the parameter values in procedures are therefore directly in these units and not in the OS units and radians which are more usual in BASIC programs.

The procedures demonstrated in *DragMeDown* plus many others are described in stages in the following sections.

Examples

Examples relevant to these procedures are named in the left margin of each page. Further information can be found in section **15 : Examples.**

3 : Program Format

In the directory *UserFiles.Programs* will be found the BASIC program *AidBlank.* This program provides a useful starting point for the users to prepare their own programs. *AidBlank* is listed below.

```
 10 REM >AidBlank
 20 REM *********************************
 30
 40 LIBRARY "<DrawAid_Lib$Dir>.Procedures"
 50 PROC_report_errors
 60 PROC_DrawAid("")
 70 PROC_finish
 80 END
 90
100 REM *********************************
110
120 DEFPROC_Objects
130 REM Place definitions of all objects here eg
140 PROC_circle(width3,black,red,100,100,20)
150 ENDPROC
160
170 REM *********************************
```

This listing shows the simplest form of program, and will draw a 20mm radius circle.

The library called at line 40 provides the main procedure **PROC_DrawAid("")**, which in turn interprets the list of objects to be drawn as determined by the user in **DEFPROC_Objects**. **PROC_circle( )** would of course be replaced by the user's own choice of object procedures.

If this program is run it will be seen that progress reports appear (briefly in this case), and the finished file is shown by the monitoring application.

The LIBRARY of *DrawAid* procedures is loaded at line 40. This statement must be present in exactly this form. The procedures **PROC_report_errors**, and **PROC_circle( )** referred to in the listing are supplied in this Library and therefore no procedure definitions for them or for **PROC_DrawAid( )** are required in the program.

Line 50 provides a simple error handler to trap any following errors. Line 60 calls the main routines of *DrawAid* which generate the *Draw* file. It must be present of course. The name to be given to the target file should be provided as a string parameter. If a

null string is provided, as shown here, the name *Untitled* is given to the file by default. There can be successive calls to this procedure each with a different name allowing multiple files to be produced.

Line 70 is a call which simply marks the last call to **PROC_DrawAid("")**, and resets various system variables. It must be included.

Line 120 declares the procedure definition **DEFPROC_Objects** where all objects to be drawn are inspected by *DrawAid*. *DrawAid* is object oriented, and each object to be drawn must be referred to inside this procedure. These objects may be defined by procedures supplied in the *DrawAid* Library, or supplied by the user, either within the program or in the user's own Library.

Objects may be of the following types, being mixed and used as required:

Standard Objects	circles, rectangles, segments etc.
Text Objects	system, outline, vector
Path Objects	move, line, arc etc.
User Objects	as defined by the user, calling *DrawAid* procedures
Sprite Objects	any sprite
CSV Objects	objects defined in CSV format

DragMeDown
MultiCircs

The **PROC_circle( )** definition in *AidBlank* is a Standard Object, and is described in the following section.

The values used as parameters in these routines can be prescribed or calculated before calling **PROC_DrawAid( )**, or calculated within **PROC_Objects** or within any user defined procedure as required. The default **units** used by *DrawAid* are **mm** and **degrees** not OS units and radians.

The standard constructs of BASIC such as FOR---NEXT, REPEAT---UNTIL, IF---THEN---ELSE etc., can be used within **PROC_Objects** to calculate the values of the procedure parameters required.

WarGame
BarCoder

Note that because *DrawAid* makes two passes through **DEFPROC_Objects**, and any procedures to which it refers, DIM statements therein will cause a re-dimension error. All DIM statements should be made prior to calling **PROC_DrawAid( )**.

Lenses
Bush

Variable values should not be assumed to be zero, as these will not be zero on the second pass. Preferably LOCAL variables should be used in all user procedures.

Graph
Table_1

If data is READ within **DEFPROC_Objects** an "out of data" error will occur unless RESTORE is placed after the last READ statement. This is required in order to reset the data pointer to the start of the DATA statements for the second pass through **PROC_Objects** which is made by *DrawAid*. when saving the *Draw* file.

WarGame Cushion Bush

If a random number generator is used, ie RND( ), then this must be seeded to produce the same "random" number on each of the two passes.

BASIC graphics commands such as MOVE, DRAW, CIRCLE, ORIGIN etc. should not be used. The more extensive *DrawAid* object procedures **PROC_circle( )** etc., must be used instead.

It is suggested that all data entry and general calculation is carried out prior to calling **PROC_DrawAid.** The results of data entry or calculations can be passed to **PROC_Objects** as global variable values.

If a BASIC statement such as I**NPUT "Enter the circle diameter ";diam** is included before calling **PROC_DrawAid( )** this will cause an *Edit* task window to open, and the request for data is made to the user. Similarly, **PRINT** statements will also appear in an *Edit* task window. At the end of the program this task will quit, and the window contents are available for handling by *Edit*. Text files recording the program's machinations can thus be generated.

The best way to learn any program is to start using it as soon as possible, and to this end many ready to run examples are included in the directory *Examples*. If problems are encountered after trying some examples, and reading this Guide, the user is advised to work through the step by step tutorial in section **17 : Tutorial**.

: Notes

4 : Standard Objects

Many parts of drawings, such as circles, arcs, and rectangles are often repeated and so a set of procedures to generate these standard shapes are available in *DrawAid*. The principal list is:

PROC_line( )	**PROC_spline( )**
PROC_triangle( )	**PROC_rectangle( )**
PROC_polygon( )	**PROC_circle()**
PROC_arc( )	**PROC_sector( )**
PROC_segment( )	**PROC_ellipse( )**
PROC_quadrant( )	**PROC_frame( )**
PROC_rounded_box( )	**PROC_plaque_box( )**
PROC_arrow_out( )	**PROC_arrow_in( )**
PROC_dimension( )	**PROC_dimension_off( )**
PROC_dimension_vector( )	**PROC_dimension_off_vector( )**
PROC_xaxis( )	**PROC_yaxis( )**
PROC_axes( )	**PROC_ring( )**
PROC_poly_spline( )	

The parameters required by the procedures vary, but are generally of the form:

(ow, oc, fc, various geometry values)

ow is the outline width in points
Several constants have been pre-defined for the user as parameters and have the following values.

thin	= 0	the thinnest line
width0	= 0	the same as thin
width1	= 1	one point wide
width2	= 2	two points wide
..... up to		
width9	= 9	nine points wide

as well as these values any other value of line width may be inserted numerically.

oc is the outline colour and fc is the fill colour
To save remembering colour numbers a variety of colour facilities are provided, and described below in the section **11 : Colours**. However, most useful are the desktop colours for values of **oc** and **fc**, and these can be referred to by the following names.

colour name	desktop colour
white	0
grey1	1

grey2	2
grey3	3
grey4	4
grey5	5
grey6	6
black	7
dark_blue	8
yellow	9
light_green	10
red	11
straw	12
dark_green	13
orange	14
light_blue	15
none	-1
transparent	-1
clear	-1

If desired the desktop colour number may be inserted as a numerical value rather than a colour name.

parameters defining various geometry values
These define specific aspects for each geometry such as the co-ordinates of reference points, the radius of an arc, or an angle, etc. These co-ordinates are given in user units, which default to **mm**, and are measured from the drawing origin, which is at the lower, left-hand corner of the *DrawAid* display window. Angles are all in **degrees**. An origin point for each object exists, usually denoted by **xo,yo**, but this is given below for each object.

Note that these parameters may be passed to the procedures either by a numeric value, or by any user defined variable name in the usual manner.

The definitions of each of the standard objects, and the parameters which they take, are listed below.

mmPaper
Compass
Trailer

PROC_line(ow,oc,xs,ys,xe,ye)
ow and **oc a**re as defined above. Note that no value for the fill colour **fc** is required. **xs,ys** defines the start of the line and **xe,ye** the end. **xs,ys** is also the origin of the line.

Sprocket

PROC_spline(ow,oc,xs,ys,xc1,yc1,xc2,yc2,xe,ye)
This object defines a single Bezier curve using two control points. The origin is at **xs,ys** and the end of the curve is at **xe,ye**. The two control points **xc1,yc1** and **xc2,yc2** have to be located to give the desired curve as in *Draw.* Note that there is no fill colour **fc** required.

PROC_triangle(ow,oc,fc,x1,y1,x2,y2,x3,y3)
x1,y1 is the origin of the triangle. **x2,y2,** and **x3,y3** are the other two vertices.

Trailer

PROC_rectangle(ow,oc,fc,xo,yo,width,height,angle)
The rectangle is initially orientated horizontally with its origin **xo,yo** at the lower left corner, and the width and height defined in the conventional way. If the value of **angle** is not zero the rectangle is rotated anti-clockwise about its **xo,yo point**, through **angle**. The value of **angle** is in degrees.

DragMeDown
BigWheel
CarWheel

PROC_polygon(ow,oc,fc,xc,yc,radius,sides,startangle)
xc,yc is at the centre of the circumscribing circle of the prescribed **radius**. The value of **sides** is an integer number not less than three. The polygon is initially orientated with the first vertex, which is the object origin, horizontally to the right of **xc,yc**, It is then rotated anti-clockwise through the value of **angle** in degrees.

Circles

PROC_circle(ow,oc,fc,xc,yc,radius)
The circle is generated from four Bezier curves, and the location of the control points has been determined to give the closest possible approximation to a pure circle. **xc,yc** is the circle centre. The object origin is the point on the circle horizontally to the right of the centre.

PROC_arc(ow,oc,fc,xc,yc,radius,startangle,endangle)
xc,yc is the centre of the arc circle. The **startangle** and **endangle** are measured anti-clockwise from the horizontal. The object origin is at the end of the arc defined by **startangle.** Note that the angle subtended by the arc is not allowed to exceed 359 degrees. It is assumed that for any larger arc you will require **PROC_circle().**

PieChart
ArcChart

PROC_sector(ow,oc,fc,xc,yc,radius,startangle,endangle)
This procedure generates a pie shaped object. **xc,yc** is the centre of the sector circle, and the object origin. The **startangle** and **endangle** are measured anti-clockwise from the horizontal. Note that the angle subtended by the arc is not allowed to exceed 359 degrees.

PROC_segment(ow,oc,fc,xc,yc,radius,startangle,endangle)
This procedure generates the lens shaped object formed between an arc and its chord. **xc,yc** is the centre of the segment circle. The **startangle** and **endangle** are measured anti-clockwise from the horizontal. The object origin is at the end of the arc defined by **startangle.** Note that the angle subtended by the arc is not allowed to exceed 359 degrees.

PROC_ellipse(ow,oc,fc,xc,yc,majorradius,minorradius,angle).
xc,yc is the geometric centre of the ellipse. The two radii can be in any order, but *DrawAid* orientates the ellipse so that initially the major axis is horizontal. This axis is then rotated anti-clockwise from the horizontal through **angle** measured in degrees. The object origin of the ellipse is the point on the ellipse horizontally to the right of the centre before rotation, i.e. at the right end of the major axis.

DragMeDown

Quadrant

PROC_quadrant(ow,oc,fc,xc,yc,radius,quadrant)
xc,yc, is the centre of the quadrant circle. **quadrant** can take the values, 1,2,3 and 4 according to standard mathematical notation.

Frame

PROC_frame(ow, oc, fc, xo, yo, width, height, wall, angle).
This procedure generates a hollow rectangular frame in the same manner as **PROC_rectangle( )** described above. The additional parameter **wall** defines the wall thickness of the frame. Only the wall is filled with colour **fc**. The interior is truly transparent.

Label
CSVGraph

PROC_rounded_box(ow,oc,fc,xo,yo,width,height,corner_radius).
This procedure generates a rectangle with corners rounded-off to **corner_radius.**

Plaque

PROC_plaque_box(ow,oc,fc,xo,yo,width,height,corner_radius). This is similar to the previous procedure, and generates a rectangle having corners fitted with re-entrant arcs of **corner_radius.** Note: **PROC_plaque( )** is also valid.

Arrows

PROC_arrow_out(ow,oc,xo,yo,length,angle)
This procedure draws an arrow-head pointing outwards from the origin point **xo,yo**. The point of the arrow is **length** from the origin and set at **angle** degrees anti-clockwise from the horizontal. No fill colour **fc** is required.

Arrows

PROC_arrow_in(ow,oc,xo,yo,length,angle)
This procedure draws an arrow-head pointing inwards to the origin point **xo,yo**. The tail of the arrow is **length** from the origin and set at **angle** degrees anti-clockwise from the horizontal. No fill colour **fc** is required.

Dimensions

PROC_dimension(xs,ys,xe,ye)
PROC_dimension_vector(xs,ys,xe,ye)
These produce a dimension line with arrow-heads between the start point **xs,ys** and the end point **xe,ye**. They place the value of the dimension adjacent to the line. The format of the dimension line and the text are governed by **PROC_dimension_style( )** and **PROC_dimension_vector_style( )** given in section **13: Other Procedures** below. The value will be positioned between the two arrow-heads, but if there is insufficient space it will be omitted. Outline text is used unless the vector form of the procedure is called in which case Vector text is used.

PROC_dimension_off(xs,ys,xe,ye,offset)
PROC_dimension_off_vector(xs,ys,xe,ye,offset)

This is identical to the previous procedure, but the dimension is off-set from the end-points. Using a negative value for **offset** will place the dimension on the other side of the line between the points.

PROC_xaxis(ow,oc,xo,yo,xmin,xmax,ticksize,tickspace)
This will generate a horizontal axis with arrow-head and ticks, between the values of **xmin**, and **xmax**, and the origin located at **xo,yo**. All values default to **mm**.

PROC_yaxis(ow,oc,xo,yo,ymin,ymax,ticksize,tickspace)
This will generate a vertical axis in the same manner as above.

AxesEx
AxesOsc

PROC_axes(ow,oc,xo,yo,xmin,ymin,xmax,ymax,ticksize,xtickspace,ytickspace)
This procedure will draw two dimensional axes with the origin at **xo,yo**, extending between **xmin** and **xmax** horizontally and **ymin** and **ymax** vertically. Legends can be positioned as required using **PROC_vector_text( )**.

Rings

PROC_ring(ow, oc, fc, xc, yc, inner_radius, outer_radius)
This generates a hollow annulus centred on **xc,yc** where only the ring between the inner and outer radii is filled with colour **fc**. Again, the interior is truly transparent.

PROC_poly_spline(ow,oc,fc,xo,yo,npoints)

PolySpline

This procedure fits a smooth curve through a line of **npoints%** where the points are contained in the array variables **polyx(i%)** and **polyy(i%)** where **i%** varies from 1 to **npoints%**. A maximum of 50 points can be stored in these arrays. Duplicate values of **polyx( )** are not permitted. The slopes at the end of the line are calculated from the rate of change of slope near the end of the line. If this results in an unacceptable slope, an additional defining point close to the end point should be inserted.

Standard Symbols

In addition to the above standard objects a number of simple symbols, centred on the point **xc,yc**, are available for use in diagrams.

Symbols
EUflag
PolyJet

PROC_symbol_circle(xc,yc)
PROC_symbol_cross(xc,yc)
PROC_symbol_diamond(xc,yc)
PROC_symbol_hspread(xc,yc)
PROC_symbol_ring(xc,yc)
PROC_symbol_square(xc,yc)
PROC_symbol_star(xc,yc)
PROC_symbol_triangle(xc,yc)
PROC_symbol_vspread(xc,yc)

The style of the symbol is defined by **PROC_symbol_style(ow,oc,fc,size)** which applies to all subsequent symbols. If this procedure is omitted from **PROC_Objects** the symbol style defaults to **PROC_symbol_style(0.25,black,red,2.5)**, where the line width **ow** is in points and the **size** is in mm.

: Notes

5 : Text Objects

Four procedures for placing text on a drawing are provided.

PROC_text( )
PROC_vector_text( )
PROC_outline_text( )
PROC_outline_text_pt( )

Table_1

PROC_text(colour,xo,yo,text$)
Provides the system font. Any of the colour names above or given in the section **11 : Colour** may be used for **colour**. **xo,yo l**ocates the lower left corner of the first character of the text. **text$** has the same scope as any normal BASIC string. The text size is prescribed to the *Draw* default values of 6.4 points wide, and 12.8 points high.

VectorText
DragMeDown
AxesOsc
Graph
BarCoder

PROC_vector_text(colour,xo,yo,size,weight,style,angle, text$)
This special *DrawAid* font is given the name **Vector**, because of the manner of its generation. The characters are defined within *DrawAid*, and are produced by line drawing, in the same manner as all other standard objects described above. The text cannot be edited within *Draw* as a normal font. However, it can be produced in various forms, can be placed at any angle, and be reproduced on a pen plotter.

colour, xo,yo, and **text$** are defined in the same manner as above. The other parameters have the following meanings and values:

size is given in mm, and refers to the height of the text capitals. Character width varies with each character.

weight is the thickness of the line with which the text is drawn. It is measured as a decimal fraction of size, and may sensibly take any numerical value less than about 0.2. However, the following names are provided to ease use. The fraction of **size** which they represent is also given.

thin	= 0	the thinnest line which can be drawn
light	= 0.05	
medium	= 0.09	
bold	= 0.13	

The user can use intermediate numerical values for other weights of line.

style refers to the slope of the characters from the vertical.
It is measured in degrees and two pre-defined names are provided.

regular	= 0	upright characters
oblique	= 12	sloping characters

The user can provide other numerical values as required.

angle refers to the orientation of the text string. It is measured in degrees. The origin **xo,yo** is located at the lower left corner of the text string, and the whole string is rotated in an anti-clockwise direction through **angle** degrees about this point.

FancyFont Plaque Table_2 Label Compass

PROC_outline_text(
fontname$,textcol%,backcol%,xo,yo,width,height,text$,angle)
PROC_outline_text_pt(
fontname$,textcol%,backcol%,xo,yo,width,height,text$,angle)
These two procedures allow access to the extensive Acorn families of outline fonts.

In both procedures **fontname$** is the name of any outline font, eg "Trinity.Bold", previously "seen" by the system, ie the *Font* application needs to have been run.

Both **textcol%** and **backcol%** (the antialiasing background colour) can assume any valid colour name or value, eg **red**.

The origin **xo,yo** is located at the lower left corner of the text string.

width and **height** refer to the dimensions of the nominal capital, and are in **mm** in the first procedure and in **points** in the second. **text$** can be any string.

angle is the rotation in the font base line in anticlockwise **degrees** from the horizontal.

If the requested outline font is not available an error message is returned. The system font is not substituted.

6 : Path Objects

Whilst it is possible to generate many complex drawings using the above standard objects, more flexibility has been provided within *DrawAid* by use of path objects.

Path objects are prepared with a set of procedures which directly generate the path elements used within *Draw*. The user is assumed to be familiar with the path nature of the Draw application, and the manner in which paths are constructed from move and draw elements.

This list of procedures available is:

PROC_new_path( )
PROC_move( )
PROC_draw( )
PROC_curve( )
PROC_move_rel( )
PROC_draw_rel( )
PROC_curve_rel( )
PROC_arc_path( )
PROC_vector_move( )
PROC_vector_draw( )
PROC_locate_path( )
PROC_rotate_path( )
PROC_scale_path( )
PROC_odd_scale_path( )
PROC_scale_path_X( )
PROC_scale_path_Y( )
PROC_shear_path_X( )
PROC_shear_path_Y( )
PROC_flip_path_X
PROC_flip_path_Y
PROC_mirror_path_X
PROC_mirror_path_Y
PROC_poly_curve_move()
PROC_poly_curve_draw()
PROC_close
PROC_close_end_path
PROC_end_path

The simplest example of use of these procedures is:

```
DEFPROC_Objects
        PROC_new_path(thin,red,none)
        PROC_move(10,10)
        PROC_draw(60,10)
        PROC_end_path
ENDPROC
```

This program segment will generate a line object 50 mm long, and the four procedure calls are equivalent to the standard object statement:

```
DEFPROC_Objects
       PROC_line(thin,red,none,10,10,60,10)
ENDPROC
```

However, path objects are very flexible and permit the construction of complex shapes and specialised procedures. In fact all of the standard procedures are defined in just this way. Section **17 : Tutorial**, gives further examples of using these procedures.

Path objects have a limit of 1000 points. If this limit is exceeded a warning is given. The problem path should be split into sections.

The definitions of each of the path procedures, and the parameters which they take, follow.

PathWay

PROC_new_path(ow,oc,fc)
Begins a new path with outline width **ow**, outline colour **oc**, and fill colour **fc**, having the same meanings as for standard object procedures. This procedure must be followed by a **PROC_move( )**, and the path must finish with either **PROC_end_path** or **PROC_close_end_path**.

PathWay

PROC_move(x,y)
If this occurs at the start of a new path it causes a move to location **x,y** and sets the path origin **xo,yo** to this point. The origin is used as a reference for other procedures such as **PROC_scale( )** and **PROC_rotate( )**. If **PROC_move(x,y)** occurs elsewhere the path line is broken and relocated at **x,y.**

PathWay

PROC_draw(x,y)
Draws a straight line from the current location to point **x,y.**

SineWave2

PROC_curve(xe,ye,xc1,yc1,xc2,yc2)
Draws a Bezier curve from the current location to the point **xe**,**ye**. **xc1**,**yc1** and **xc2**,**yc2** are the control points related to the start point and end point of the curve respectively.

PathWay Mirror

PROC_locate_path(xo,yo,xposition,yposition)
Shifts all of the points on the path so that the point **xo,yo**, is relocated at **positionx, positiony**. Point **xo,yo,** may be chosen to be the path origin, or any other point on the path, or indeed any other point not on the path.

PathWay

PROC_rotate_path(xo,yo,angle)
Rotates all of the points on the path about the point **xo,y**o anti-clockwise through **angle** degrees. Point **xo,yo** may be chosen to be the path origin, or any other point on the path, or indeed any other point not on the path.

WarGame

PROC_scale_path(factor)
Magnifies the size of the path object by a **factor**, which may be smaller or greater than one. Only the origin point of the path remains in its original position.

PROC_scale_path_X(factor)
This allows scaling of the path defined to that point in the X or horizontal direction only. Scaling takes place relative to the path origin (the first point on the path) which remains unmoved.

PROC_scale_path_Y(factor)
Permits scaling in the Y or vertical direction.

PROC_odd_scale_path(xo,yo,factorx,factory)
Magnifies the size of the path unequally in both x and y directions by **factorx** and **factory**. The scaling takes place about point **xo,yo,** which may be the path origin point, or any other point on the path, or indeed any point not on the path.

PROC_shear_path_X(angle)
PROC_shear_path_Y(angle)
These procedures shear all points on the path through an **angle**. Shearing takes place relative to the graphics origin. If shearing is required about some other point, such as the path origin, then the path should first be located with that point at the origin using **PROC_locate( )**. The reference point can then be re-located where required. When sequentially using these two procedures they produce a warping effect.

PROC_flip_path_X
PROC_flip_path_Y
These allow the path, which has been defined up to that point in the program, to have its X or Y values flipped about the axis through the path origin but not copied.
i.e. **PROC_flip_path_X** will flip the path X values about the vertical axis. There are no parameters passed.

PROC_mirror_path_X
PROC_mirror_path_Y

Mirror Lenses

These will mirror the path defined at that point in the program in X or Y values about the axis through the path origin. i.e. **PROC_mirror_path_X** will copy the X values of the path in reverse to the other side of the vertical axis. Again, no parameters are passed.

PROC_move_rel(dx,dy).
This is a path procedure proscribing a move from the current **x,y** location to a point relatively located **dx,dy** away.

Lenses

PROC_draw_rel(dx,dy).
In the same manner this procedure will draw to a new point incrementally removed by **dx,dy**.

PROC_curve_rel(dxe,dye,dxc1,dyc1,dxc2,dyc2).
This will draw a Bezier curve where the end point **dxe,dye**, and the two control points **dxc1,dyc1**, and **dxc2,dyc2** are positioned relative to the start point by the values given.

PROC_arc_path(xc,yc,angle).

Lenses

This procedure will include a circular arc element on the path starting at the current **x,y** location, centred on location **xc,yc**, and then extending through **angle** degrees. **angle** represents an anti-clockwise arc, and negative values will give a clockwise arc. Arc elements are limited to 90 degrees each.

PROC_vector_move(l,angle).
Allows a move from the current **x,y** location to a new point located at distance **l** and **angle** degrees removed. The positive value of **angle** is measured anti-clockwise from the horizontal.

PROC_vector_draw(l,angle)
offers the same facility for drawing a line element.

PROC_poly_curve_move(npoints%)
PROC_poly_curve_draw(npoints%)

PolyCurve
PolyJet

These procedures fit a smooth curve through a line of **npoints%** where the points are contained in the array variables **polyx(i%)** and **polyy(i%)** where **i%** varies from 1 to **npoints%**. A maximum of 50 points can be stored in these arrays. Duplicate values of **polyx( )** are not permitted. The first procedure moves to the first point in the array, whilst the second procedure draws to the first point.

PathWay
SineWave1

PROC_end_path
completes the path. Although the start and end points may be co-located they can be moved independently within *Draw*, and the path is said to be open. Clearly a truly open object should terminate in this way.

PathWay

PROC_close
This procedure will close the path but not end it. It must be followed by **PROC_end** or **PROC_move( )**, producing a path with two or more closed sections connected by an invisible move.

Mirror

PROC_close_end_path
completes the path, and joins the first and last points, which should be co-incident. This procedure should be used for closed shapes. Either **PROC_end_path** or **PROC_close_end_path** is required as the last entry in a path sequence.

For a further example of using path objects see the following section and section **17 : Tutorial**.

Note: As arcs are constructed in *Draw* files from cubic splines, their shape can only approximate to a true circular arc. The actual shape depends on the position of the two control points. The user may find that the location of an expected intercept with such an arc may be in obvious error on the drawing. The solution is to split the arc into two parts with the intercept point being common to the two parts. This forces the curve through the desired point.

7 : User Objects

Using the supplied path procedures the user can define his own frequently used range of objects. These can then be stored in a Library and called as required. A physics teacher, for instance, may require to produce drawings of sets of lenses and he could define his own procedure e.g.

Lenses

PROC_lens(ow,oc,fc,xo,yo,d,t,r1,r2)
xo,yo locates the lens reference origin i.e. on the axis at the left surface. **d** is the lens diameter, **t** is its edge thickness, **r1**, and **r2** are the surface radii.

The two-part listing below illustrates a program defining and using **PROC_lens( )**. The lens train produced is shown in the diagram. After proving the procedure, it would be stored in a User Library, thereby reducing the program to only 17 lines.

```
REM >Lenses

LIBRARY "<DrawAid_Lib$Dir>.Procedures"
PROC_report_errors
PROC_DrawAid("LensSet")
PROC_finish
END

DEFPROC_Objects
REM Place definitions of all objects here eg
PROC_lens(width2,black,grey1,70,60,100,15,300,-300)
PROC_lens(width2,black,dark_blue,102,60,100,5,-200,-300)
PROC_lens(width2,black,grey1,110,60,100,5,-300,200)
PROC_dot_dashed
PROC_line(width3,black,55,60,140,60)
ENDPROC
```

cont.

cont.

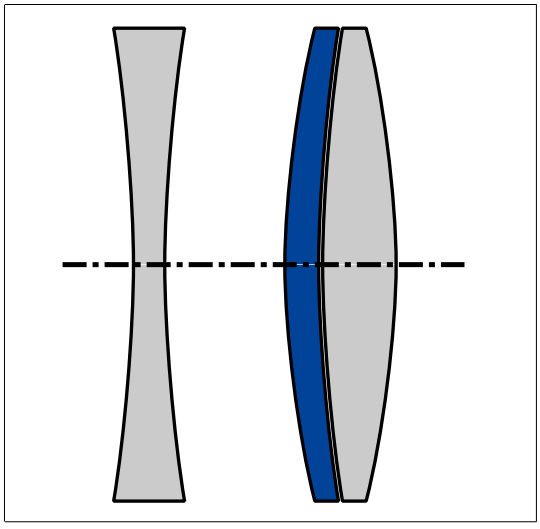
```
DEFPROC_lens(ow,oc,fc,xo,yo,d,t,r1,r2)
REM xo , yo is the location of the left surface centre
REM d is the lens diameter, t is the edge thickness
REM r1 and r2 are the surface radii
REM a1 and a2 are the semi-conical angles of the surfaces

LOCAL r,a1,a2,x1,x2,y1,y2
r=d/2
a1=DEG(ASN(r/r1)):a2=DEG(ASN(r/r2))
x1=-r1:y1=0
x2=t-r2*COS(RAD(a2))-r1*(1-COS(RAD(a1))):y2=0

PROC_new_path(ow,oc,fc)
PROC_move(0,0)
PROC_arc_path(x1,y1,a1)
PROC_draw_rel(t,0)
PROC_arc_path(x2,y2,-a2)
PROC_move(0,0)
PROC_mirror_path_Y
PROC_locate_path(0,0,xo,yo)
PROC_close_end_path

ENDPROC
```

saved in user Library



LensSet

8 : Sprite Objects

Sprite objects can be selected from a file, and placed at any location on a drawing. The required file of sprites is saved in the directory *SpriteFile*, and the file can contain any number of individually named sprites. In the example below the single sprite used is called *tile_1*, and is selected from the sprite file *SpriteFile.Tiles*.

```
REM >Delft
LIBRARY "<DrawAid_Lib$Dir>.Procedures"
PROC_report_errors
PROC_load_sprites("Tiles")
PROC_DrawAid("Delft")
PROC_finish
END

DEFPROC_Objects
PROC_scaled_sprite("tile_1",50,20,50,50)
PROC_flip_sprite_X("tile_1")
PROC_sprite("tile_1",55,70,0.35,20)
PROC_scaled_sprite("tile_1",100,20,20,100)
ENDPROC
```

The procedure **PROC_load_sprites(sprite_file_name$)** is placed in the program prior to the call to **PROC_DrawAid("")**. The actual sprite objects are defined within **PROC_Objects** by the calls **PROC_sprite( )**, and **PROC_sprite_scaled( )**.

Also provided are procedures flipping the sprite about its X and Y axes. A flip about the X axis is shown in the example above. These procedures should be set immediately prior to calling **PROC_sprite( ),** or **PROC_scaled_sprite( )**. The sprite is returned to its normal mode after each call to **PROC_sprite( ),** or **PROC_scaled_sprite( )**.

Delft
WarGame
RigChart

PROC_load_sprites(sprite_file_name$)
The required sprites are loaded as the file **sprite_file_name$** immediately prior to a call to **PROC_DrawAid("")**. The file **sprite_file_name$** must be present in the private directory *SpriteFile*.

Delft
WarGame
RigChart

PROC_sprite(spritename$,xo,yo,factor,angle)
This procedure will plot the sprite **spritename$** from the file **sprite_file_name$** with its lower left corner at the location **xo**,**yo**. The sprite will be plotted at **factor** times its "natural" size inclined at clockwise **angle** degrees to the horizontal. If **factor** =1 and **angle** = 0 the sprite will be plotted normally at its "natural" size.

Delft

PROC_scaled_sprite(spritename$,xo,yo,width,height)
This procedure will plot the sprite **spritename$** from the file **sprite-file-name$** with its lower left corner at the location **xo**,**yo**. The sprite will be scaled to fit the rectangle of size width x height, where these dimensions are user units, but default to mm.

Delft Wargame

PROC_flip_sprite_X(spritename$)
After this using procedure **PROC_sprite( )** or **PROC_scaled_sprite( )** will plot the sprite with its X values flipped. After plotting the sprite is restored to its original orientation.

PROC_flip_sprite_Y(spritename$)
After using this procedure **PROC_sprite( )** or **PROC_scaled_sprite( )** will plot the sprite with its Y values flipped. After plotting the sprite is restored to its original orientation.

PROC_flip_sprite_XY(spritename$)
After this using procedure **PROC_sprite( )** or **PROC_scaled_sprite( )** will plot the sprite with both its X and Y values flipped. After plotting the sprite is restored to its original orientation.

Note: When using **PROC_load_sprites(sprite_file_name$)** all of the sprites in that file are loaded, but only those required are saved to the *Draw* file. If you are short of memory, make sure that only the sprites you need are in **sprite_file_name$**.

9 : CSV Objects

It is sometimes convenient to prepare drawing data using other software, possibly a wordprocessor or a spreadsheet, and on some other computer system. If this data can be stored as a file of type *text* in CSV (comma separated value) format, it can be presented to *DrawAid* as one or more CSV objects. The text file is stored in the private directory *CSV* and is called from within **DEFPROC_Objects** by the following procedure.

CSV-Example CSVGraph

PROC_CSV(filename$)
filename$ is name of CSV text file in *CSV* directory.

In the following example the BASIC program is shown on the left, and the CSV file on the right. This example shows how the BASIC program provides a fixed "shell" to variable graphical data read from a CSV text file.

This facility has considerable scope for fertile imaginations. Even the spread-sheet or word-processing document can be a "shell' or "template" into which non-programming support staff can enter numerical data only.

```
REM  >CSVGraph
LIBRARY "<DrawAid_Lib$Dir>.Procedures"
PROC_report_errors
PROC_DrawAid("Graph")
PROC_finish_task
END

DEFPROC_Objects
PROC_rounded_box(width1,black,black,32,28,150,100,5)
PROC_rounded_box(width1,black,straw,30,30,150,100,5)
PROC_vector_text(black,40,60,5,medium,oblique,90,"oxygen%")
PROC_vector_text(black,90,40,5,medium,oblique,0,"hours")
PROC_axes(width1,black,50,50,0,0,100,60,3,10,5)
PROC_CSV("CSVGraph")
ENDPROC
```

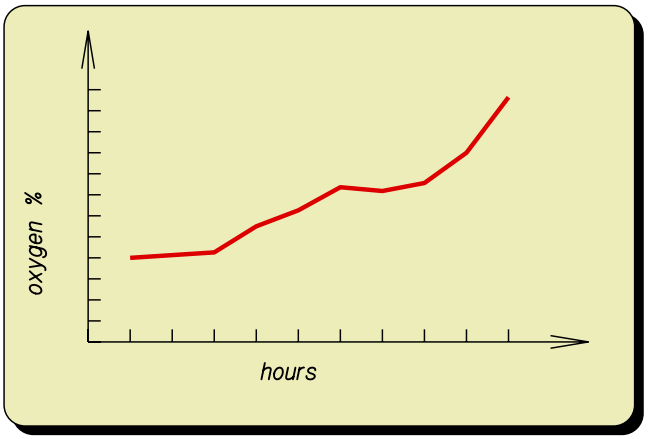
BASIC file *CSVGraph*

Text file *CSVGraph*

```
REM CSV data for graph

new_path,3,11,12
move,     10,      20
draw,     30,      21.3
draw,     40,      27.5
draw,     50,      31.3
draw,     60,      36.8
draw,     70,      35.9
draw,     80,      37.8
draw,     90,      45.0
draw,     100,     58.2
locate_path,0,0,50,50
end_path
```

The resulting *Draw* file of the graph generated is shown below.



The information in the CSV file generates a single path which is the graph line. The outline rounded box and axes information is provided by the BASIC program. Multiple lines could be stored as separate paths in the same CSV file. See the extended version of this program *CSVGraph* in the *Examples* directory. Note that the CSV names, format, and parameters taken are identical to those of the BASIC object procedures. A full list is given below, together with a three letter mnemonic which can be used in place of the full name, and is useful in spread-sheets.

Comments may be inserted anywhere in the CSV file. Lines starting with REM or the | character are not inspected for data. Spaces are ignored. Adjacent commas will cause a zero to be read.

a) Standard Objects

circle	**cir**
ring	**rin**
ellipse	**ell**
arc	**arc**
segment	**seg**
sector	**sec**
quadrant	**qua**
rectangle	**rec**
frame	**fra**
rounded_box	**rob**
plaque_box	**pla**
triangle	**tri**
polygon	**pol**
line	**lin**
spline	**spl**

b) Path Objects

new_path	**new**
move	**mov**
draw	**dra**
curve	**cur**
arc_path	**arp**
move_rel	**mor**
draw_rel	**drr**
curve_rel	**crr**
vector_move	**vem**
vector_draw	**ved**
scale_path	**scp**
odd_scale_path	**osp**
scale_path_X	**scx**
scale_path_Y	**scy**
shear_path_X	**shx**
shear_path_Y	**shy**
flip_path_X	**flx**
flip_path_Y	**fly**
mirror_path_X	**mix**
mirror_path_Y	**miy**
rotate_path	**rot**
locate_path	**loc**
close	**clo**
close_end_path	**cle**
end_path	**end**

: Notes

10 : Use of Groups

One of the advantages of D*raw* is its ability to group objects together and then to manipulate them as a whole. Because *DrawAid* can contain loops producing multiple objects, a facility to generate groups is included. Two procedures are provided.

Rings
Compass
Table_2
Trailer

PROC_new_group
PROC_end_group

The use of groups is shown in the demonstration program *DraGmEdown*, but is also illustrated by the following program fragment:

```
500 PROC_rectangle(10,10,100,100,0)
500 PROC_new_group
510           FOR radius =50 TO 20 STEP -10
520           PROC_circle(width1,red,yellow,70,70,radius)
530           NEXT radius
540 PROC_end_group
```

On un-grouping the resulting *Draw* file the concentric circles will be found to be grouped together with the square as a separate object.

Only one group may be open at one time. This means that groups cannot be nested but all groups, and any single objects at this level, are grouped together as a single object in the final *Draw* file. These two levels of nesting are more than adequate for most applications.

Note that in **Vector** text strings of more than one character the characters are grouped automatically, and such text can not be grouped further. If this is attempted a warning is given.

: Notes

11 : Colours

The following names may be used where a colour value is required:

a) desktop palette

desktop colours	desktop colour number
white	**0**
grey1 - grey6	**1 - 6**
black	**7**
dark_blue	**8**
yellow	**9**
light_green	**10**
red	**11**
straw	**12**
dark_green	**13**
orange	**14**
light_blue	**15**
none	**-1**
transparent	**-1**
clear	**-1**

b) 16 greys

g**reyscale0 - greyscale16**
greyscale16 is identical to **greyscale15** ie black

These names define 16 grey colours in the *Draw* file. However, only 8 will show on the normal desktop palette. Use the grey palette provided in the Tools directory to show all 16 greys. 16 greys will show on the printer. Select dithered output on the printer driver for a smooth gradation in level.

Additionally, the same scale is obtained from;

greyscale(i%) where **i%**=0 to 16. This allows the greys to be selected by a variable **i%**, and incremented in loops.

c) 256+ colours

Up to 256 colours may be selected at any time from a palette block by:

GreyScale
ColourScale
Cushion

rgb(i%)
where **i%**=1 to 256

i% selects the colour previously set in the palette block by:

PROC_set_rgb(i%,r,g,b)
where **i%**=1 to 256
and where values of **r, g,** and **b** range from 0 to 100, i.e. percentage saturation, and may assume fractional values eg 12.5.

Alternatively:

PROC_set_rgb_hex(i%,r%,g%,b%) may be used
where **i%**=0 to 256
and where **r%, g%,** and **b%** are integers with a maximum value of 255 decimal or &FF.

These procedures may be used wherever required to redefine the 256 colours in the palette block. Since the r,g,b definitions of the 256 colours in the palette block can be redefined as the drawing is generated, this facility allows the definitions of up to 16 million colours to be present in one *Draw* file.

Note that, whilst the *Draw* file will hold the precise colour definition, the colour displayed will depend on the screen mode. Even the 256 colour modes will show only the nearest colour which that mode allows. However, *Draw* with RISC OS 3.1 will dither the mode colours to give the best approximation of the defined colour. Note that neither the old versions of *Draw* nor *DrawPlus* will show dithered colours.

The evolving high capability colour printers and graphics screen enhancer plug-in cards for use with the earlier Acorn RISC OS machines may give a more faithful reproduction, depending on their colour ranges and the software drivers used. The extended capability of the Acorn RISC PC range of computers will automatically show the extended colour range.

12 : System

The *DrawAid* procedure Library is selected early in the main program by:

LIBRARY“<DrawAid_Lib$Dir>.Procedures”

BigWheel
CarWheel
Gears

If users require their own Library, named for instance *MyLib*, then this should be placed in the directory *DrawAid.Library* and would be selected by including:

LIBRARY“<DrawAid_Lib$Dir>.MyLib”

Multiple libraries can be stored and selected by this call as required for any particular program.

The following procedures govern the operation of *DrawAid* and must only be used in the main program, and not in **PROC_Objects**, or any user defined procedure. The error report should be included before calling **PROC_DrawAid( )**.

DragMeDown

PROC_report_errors	provides simple error report
PROC_DrawAid(filename$)	use with new programs : Note **""="Undefined"**
PROC_finish	must follow the last **PROC_DrawAid( )**
PROC_message(message$)	allows the user to send progress reports to the dialogue box. **message$** is cropped to the leftmost 47 characters.

Compass

PROC_fatal_error(message$)	allows the user to send an error report to the dialogue box. **message$** is cropped to the leftmost 47 characters.
PROC_no_hourglass	use at the start of **PROC_Objects** to leave the mouse pointer unaffected. This allows you to carry on with other tasks uninhibited while the *DrawAid* task continues in the background.

BarCoder
GearTrain

data$=FN_fetch_data(message$)	allows the user program to fetch data entered through the dialogue box. **message$** is the request sent, and **data$** is the reply. **data$** is restricted to 19 characters, and must be evaluated by the user's own routines.

Circles

PROC_set_units(<units>**)**	Use at the start of **PROC_Objects** to select units other than the default **mm** eg **feet** or **cm** etc.

Note: In the display window *DrawAid* will scale the drawing to fit the screen, unless the object is small enough to fit "full size". In this case it will be drawn at "full size". In practice this means that drawings which are either wider than about 120 mm or higher than 90 mm will be scaled down to fit the window.

13 : Other Procedures

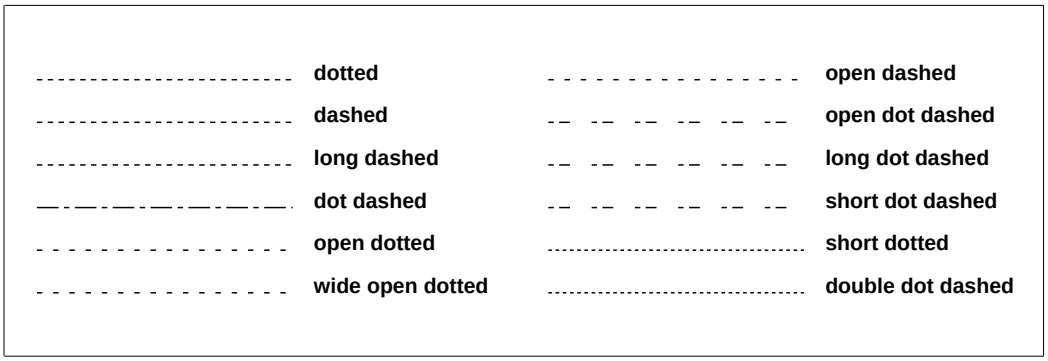
a) Line Styles

The following procedures can be used anywhere within **PROC_objects**, and will set the line style for the immediately-following, path or standard object. These styles do not affect vector text or dimension text characters. After each object the line returns to un-patterned. No parameters are passed by these procedures.

Dimensions
LineTypes
Lenses

PROC_short_dotted
PROC_dotted
PROC_open_dotted
PROC_wide_open_dotted
PROC_dashed
PROC_long_dashed
PROC_short_dot_dashed
PROC_dot_dashed
PROC_long_dot_dashed
PROC_open_dot_dashed
PROC_double_dot_dashed
PROC_open_dashed

The example called *LineTypes* in the *Examples* directory has been run to produce the figure below. This shows what the line patterns look like.



LineTypes

b) Control of caps and joints in line style use

The style of the start cap, joints and end caps used by a line can be redefined using the following procedure.

PROC_line_joints(start%,join%,end%)

The values of the three parameters have been predefined in *DrawAid* as the variables

butt	value = 0
mitre	value = 0
round	value = 1
bevel	value = 2
square	value = 2
triangular	value = 3

The values given are those defined in Acorn's *Draw* file specification. At present there is no control offered over the proportions of the arrowhead when using triangular start or finish to the line.

Instead of having to remember the values the programmer inserts the variable name. Thus for a line staring with a butt, with round joints, and a round end use:

LineJoints Frame

PROC_line_joints(butt,round,round)

The procedure is used in the same manner as the line styles procedures such as **PROC_dotted** as described above. The default values are **(butt,mitre,butt)**

c) dimensions

PROC_dimension_style(linewidth,colour,fontname$,size,places)

Dimensions

This procedure sets the format of subsequent dimensions using outline fonts as created by **PROC_dimension( )**, or by **PROC_dimension_off( )**, as described in the section **4: Standard Objects**. **linewidth** refers to the dimension line thickness, whilst **size,** defines the nominal character size in mm. The variable **places** refers to the number of places after the decimal point in the dimension value. The last place is rounded. The **colour** selected affects both the dimension line and its value. The font selected by **fontname$** is any valid font which has been "seen" by the system. If this procedure is not called the default values given below are used.

PROC_dimension_style_reset

This procedure resets the dimension style to the default values of **:**

PROC_dimension_style(0.25,black,"Homerton.Bold",4,2)

PROC_dimension_vector_style(width,colour,size,weight,style,places)

This procedure sets the format of the following dimensions using vector text as created by **PROC_dimension_vector( )**, or by **PROC_dimension_off_vector( )**, and as described in the Section **4: Standard Objects** above. **width** refers to the dimension line thickness, whilst **size, weight**, and **style** describe the dimension text and can take the same values as used for **PROC_vector_text( )**. The variable **places** refers to the number of places after the decimal point in the dimension value. The last place is rounded. The **colour** selected affects both the dimension line and its value.

PROC_dimension_style_reset

This procedure resets the dimension style to the default values of **:**

PROC_dimension_style(0.25,black,4,medium,regular,2)

: Notes

14 : DrawAid Variables

a) default variables

DrawAid is arranged with default values for a number of variables, but these can be changed within **PROC_Objects** by the user. The following list gives the default values and possible changes.

variable	default	alternatives
filename$	**"Untitled"**	<as user decides>
units	**mm mms**	units set by :- **PROC_set_units()** **cm cms** **m metre metres** **in inch inches** **ft feet foot**

```
DEFPROC_objects
  REM redefine the working units
  units=inches
  PROC_circle(width2, black, red, 3,3,2)
ENDPROC
```

The example fragment above will draw a circle 2 inches radius located 3 inches in both x and y from the origin.

If data is not to be entered in the default mm then a definition of units will normally be made at the start of **PROC_Objects** using **PROC_set_units().** see section **12: System** A **PROC_set_units()** definition inserted into the user's program before calling **PROC_DrawAid( )** will be ignored, and the default **mm** will be assumed..

b) reserved variables

There are many procedures within the *DrawAid* Library, including those described above, and wherever possible variables used by these procedures have been declared LOCAL to them. However, some GLOBAL variables can be accessed by the user, and some care is required in their use.

DrawAid operates by making two passes over the procedure **PROC_Objects**. It does this from within **PROC_DrawAid( )**. Many of the variables used within **PROC_DrawAid( )** are therefore GLOBAL with respect to **PROC_Objects**. Consequently changes to the values of these variables may affect the running of the combined program. Some of these variable names are therefore RESERVED and must not be used. However, all of these reserved variable names end with the two characters, underscore, and capital **A**, ie "**_A**", and are unlikely to be duplicated by meaningful names declared by the user. Nevertheless, such endings should be avoided. The ending "**_A**" has also been appended to the non-user procedures in the Library for the same reason.

In addition to these reserved variables there are a number of GLOBAL predefined constants, available for convenience, which have been referred to above. These can be used as required but should not normally be redefined. A summary list of these names is given below.

units, mm, mms, cm, cms, m, metre, metres, in, inch, inches, ft, feet, foot

white, grey1 - grey6, black, none, transparent, clear

greyscale0 - greyscale16, greyscale( ), rgb( )

dark_blue, yellow, light_green, red, straw, dark_green, orange, light_blue

light, medium, bold, regular, oblique

thin, width0 - width9

butt, mitre, round, bevel, square, triangular

filename$

The names **filename$,** and **units** may be redefined according to their use described in section **13 : Other Facilities**.

15: Memory and other Facilities

a) Memory management

The amount of memory claimed by *DrawAid* when it starts up is only 32k. This is sufficient for ancillary functions such as opening a task window and dumping files as described below. It means that the application can sit quiescent on the icon bar without demanding much memory. When a *Draw* file is required to be shown the required memory will be extracted from the free pool and released again after use.

If a *DrawAid* BASIC file is dragged into *DrawAid* then a subtask is started which requires a minimum of 160k plus the length of any sprite files to be loaded. The default setting is 320k which will therefore manage drawings using source sprite files of up to 160k. The length of the file generated is only dependent on the free disc space available.

If large sprite files are to be used then more memory may need to be made available to the subtask by changing the 320k in the application file *SetMaxMem* to a suitable value.

If no sprites are used then the sub task memory requirement may be reduced to 160k.

If you have a particularly large file or require the maximum speed you can run your BASIC program without having the *DrawAid* application running, i.e. <Quit> from the icon bar. Provided that *DrawAid* has previously been on the icon bar, and the machine has not been reset, the *DrawAid* "environment" will be remembered. In this event the BASIC program will single task, and will be allocated the total memory showing in the *TaskManager* Next slot. This Next slot memory allocation may need to be adjusted to accommodate any large sprite files to be loaded.

b) Inspecting files

If any file is dragged to the *DrawAid* icon whilst the <Alt> key is being held down, a hexadecimal "dump" of the file is produced and stored in the same directory as the source file. This can then be inspected by loading the dumped data file into *Edit*.

c) Launching BASIC programs

Not just *DrawAid* user programs, but any BASIC program can be started as a task by dragging it onto the *DrawAid* icon. This is useful if a long background calculation is required. PRINT **or** INPUT statements in the BASIC program will cause a task window to be opened, which will record information for later use.

d) Opening a BASIC window

If The <Shift> key is held down whilst clicking on the *DrawAid* icon, a BASIC task window is opened for input. This is useful for instant BASIC sums, but operating system commands also, such as ***show** or ***status,** can be entered here without leaving the desktop. The command ***gos** will force a change to the operating system supervisor.

e) Resetting the dialogue box

If a subtask is Quit from the Task Manager before it has completed then the dialogue box may be left displaying redundant information. This can be removed by clicking with Select on the icon bar icon. This action also opens *DrawAid.UserFiles* and *DrawAid.UserFiles.DrawFiles* directory viewers, and places them adjacent to the dialogue box. If these directory viewers are not required then clicking with Adjust will reset only the dialogue box.

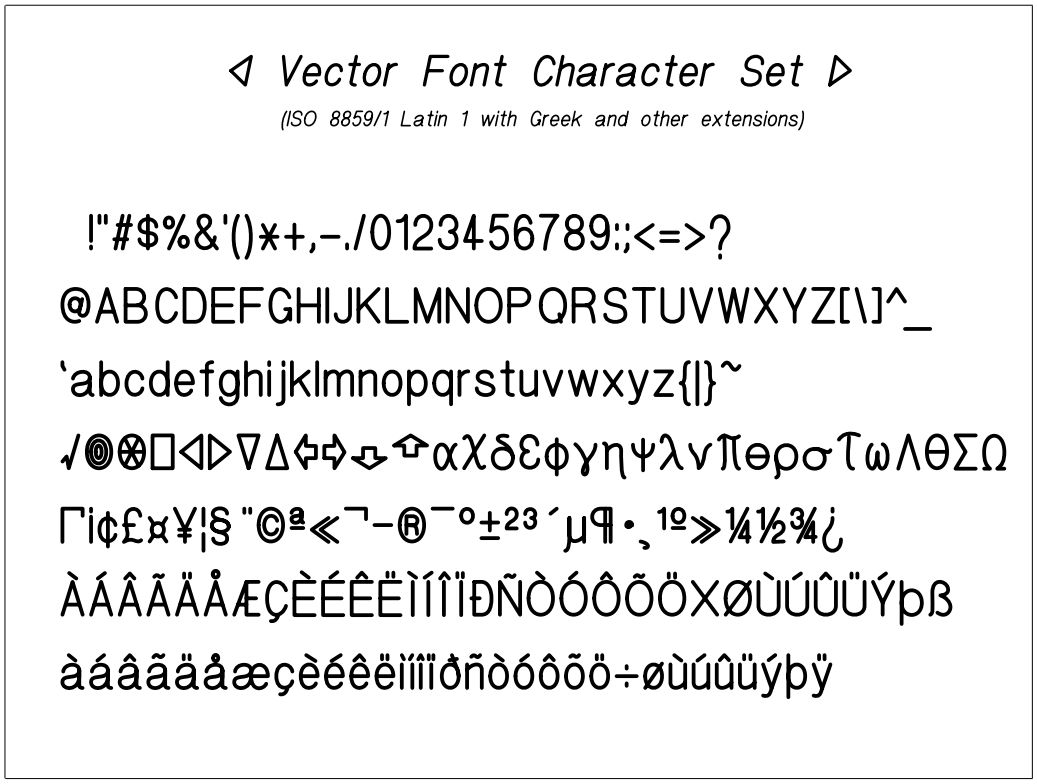
16 : Example Programs

An extensive range of example programs is included in the directory *UserFiles.Examples*. These cover use of most of the procedures described in the sections above, and give a good indication of the flexibility of *DrawAid*. If the examples are run the resulting *Draw* files will show the complexity of drawings which can be produced. In the main text, appropriate examples from this directory have been indicated in the left margin. The user should note that this directory also includes the directory *Tutorial*, and further examples which are described in section **17 : Tutorial**.

: Notes

17 : Vector Font

The resident *DrawAid* font **Vector** can be included in a drawing. It can have any size, weight, and obliqueness, and can be set at any angle. The font is ideal for output to line plotters as it will provide an exact copy of the screen, and not be dependant on the plotter's character set. The table below shows all of the characters of the font.



: Notes

18 : Tutorial

To get the best learning experience from the following programs perhaps they should be typed in and run. However, if this does not appeal they will all be found ready and lurking in the directory *UserFiles.Examples.Tutorials*.

a) Take a Blank Sheet of Paper

Well, not exactly blank. The BASIC program *AidBlank* is provided as a ready starting point for most programs using the *DrawAid* procedures. If you have not already done so, read Section **3 : Program Format** before progressing further with the tutorial. Section 3 describes the main features of *AidBlank*, which is reproduced here, renamed *TutorialA*, as the starting point for this introductory tutorial.

```
 10 REM >TutorialA
 20 REM ***********************************
 30
 40 LIBRARY "<DrawAid_Lib$Dir>.Procedures"
 50 PROC_report_errors
 60 PROC_DrawAid("tutorial_a")
 70 PROC_finish
 80 END
 90
100 REM ****************************************
110
120 DEFPROC_Objects
130 REM Place definitions of all objects here eg
140 PROC_circle(width3,black,red,100,100,50)
150 ENDPROC
160
170 REM ****************************************
```

Line 40 loads the required Library of *DrawAid* procedures. RISC OS is made aware of the location of this library when the *DrawAid* icon is double-clicked upon with <select>. Line 50 provides a simple error handler. Line 60 calls the main Library procedure **PROC_DrawAid( )**, and this is followed by **PROC_finish** and the **END** of the main program. All of the objects to be drawn and saved as *Draw* files are defined within the procedure definition of **PROC_Objects** forming the rest of the listing.

After double-clicking on the *DrawAid* icon to install the application on the icon bar, all that is necessary to run any program is to drag the BASIC program to this icon, and the required *Draw* file will be produced, and saved.

b) Draw a Circle

In the case of *AidBlank* the objects are restricted to one circle which is defined using the *DrawAid* procedure **PROC_circle(width2,black,red,100,100,50)**. The circle will be drawn with a black line, width of 2 points, filled with red, its centre will be located 100mm from the lower left hand corner of the drawing both horizontally and vertically. It will be of radius 50mm. The name *tutorial_a* has been given to the resulting *Draw* file by passing it as a string parameter in **PROC_DrawAid("tutorial_a")** in line 60. Leaving this parameter as a null string i.e. "" saves the drawing with the default name of *Untitled*.

c) Draw a Circle of Holes

One of the most common items in engineering drawings is a set of holes drilled on a prescribed pitch circle. This is rather difficult if not impossible to prepare using the mouse and *Draw*. However, by use of a FOR...NEXT loop we can modify our example to prepare a group of such holes. The modified program *TutorialB*, is shown below in the two-part listing.

```
 10 REM  >TutorialB
 20 LIBRARY "<DrawAid_Lib$Dir>.Procedures"
 30 PROC_report_errors
 40 PROC_DrawAid("tutorial_b")
 50 PROC_finish
 60 END
 70
 80 DEFPROC_Objects
 90 hole_size=11   :REM hole 11mm diameter
100 radius=hole_size/2
110 pcr=50         :REM pcr=pitch circle radius
120 noh=8          :REM noh=number of holes
130 xcentre=100
140 ycentre=100
150 FOR hole=1 TO noh
160 angle=hole*2*PI/noh+PI/noh
```

cont.

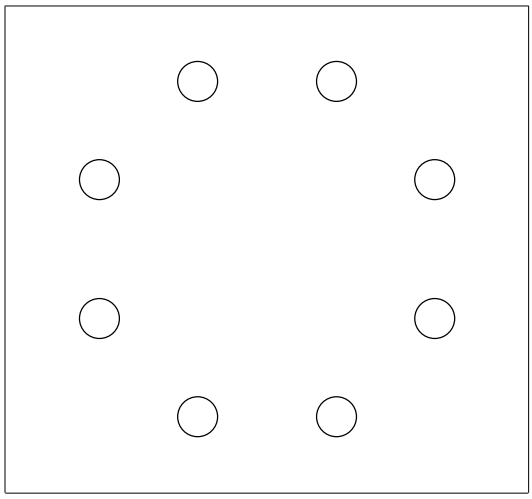
cont.

```
170 xc=xcentre+pcr*COS(angle)
180 yc=ycentre+pcr*SIN(angle)
190 PROC_circle(width1,black,none,xc,yc,radius)
200 NEXT hole
210 ENDPROC
```

This time, a number of variables have been introduced to specify the location of the centre of the ring of holes **xcentre** and **ycentre**, the pitch circle radius **pcr**, the number of holes **noh**, and the diameter of the holes **hole_size**. These variables allow the program to produce pitch circles of any size, and number of holes, by a change of values which could be entered through an INPUT statement before calling **PROC_DrawAid( )**.

The FOR...NEXT loop between lines 150 and 200 calculates the centre of each hole in turn, and draws a circle at that point using the same procedure **PROC_circle()** as above.

You should prepare this program, and run it as before to generate the *Draw* file named *tutorial_b*. As *DrawAid* groups all of the objects it prepares, it will be found that the eight holes are grouped together within the *Draw* file.



tutorial_b

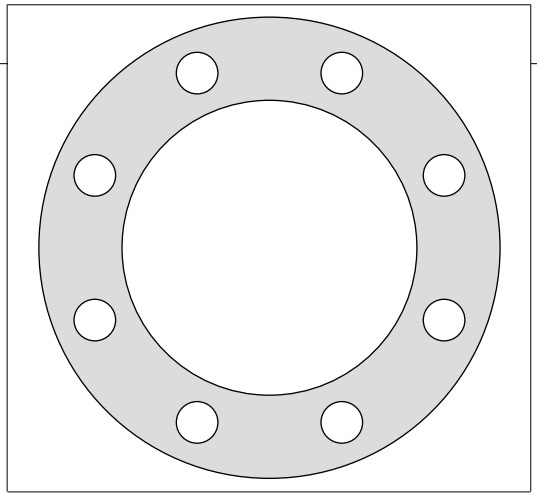
d) Drill a Flange

All of the hole "objects" generated above were grouped together by *DrawAid*, and this group would have included any other objects which might have been defined. However, it is possible to have one level of sub-grouping, as might be desired for a bolted flange, where the holes would form a sub-group. *TutorialC* lists the modifications necessary to do this.

Two new circles, representing the inner and outer radii of the flange, are defined between lines 140 and 170. The bolt holes are then grouped together by introducing **PROC_new_group** at line 180. The group must be ended by **PROC_end_group** when all of the holes have been defined, ie at line 250. On running this program the resulting *Draw* file main group will contain two circle objects plus one group of holes.

```
 10 REM  >TutorialC
 20 LIBRARY "<DrawAid_Lib$Dir>.Procedures"
 30 PROC_report_errors
 40 PROC_DrawAid("tutorial_c")
 50 PROC_finish
 60 END
 70
 80 DEFPROC_Objects
 90 hole_size=11   :REM hole 11mm diameter
100 radius=hole_size/2
110 pcr=50         :REM pcr=pitch circle radius
120 noh=8          :REM noh=number of holes
130 xcentre=100:ycentre=100
140 outer_radius=pcr+hole_size
150 inner_radius=pcr-hole_size
160 PROC_circle(width1,black,grey1,xcentre,ycentre,outer_radius)
170 PROC_circle(width1,black,white,xcentre,ycentre,inner_radius)
180 PROC_new_group
190 FOR hole=1 TO noh
200 angle=hole*2*PI/noh+PI/noh
210 xc=xcentre+pcr*COS(angle)
220 yc=ycentre+pcr*SIN(angle)
230 PROC_circle(width1,black,white,xc,yc,radius)
240 NEXT hole
250 PROC_end_group
260 ENDPROC
```

Note that the central hole and bolt hole fill colour has been made white to obscure the flange colour. True transparency is possible using path procedures. See the *Examples* directory.



tutorial_c

e) Bolt a Flange

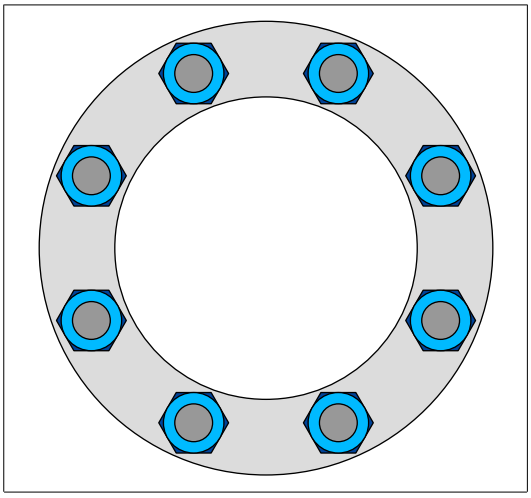
In the above example a group was defined outside the FOR...NEXT loop. The procedures could equally well have been used to group objects inside the loop. The program *Tutorial_D* shows how this is done, by defining group objects representing nuts bolted to the earlier flange.

Each nut is defined by a hexagon at line 230, and two circles at lines 240 and 250. These three items are grouped together by the procedures at 220 and 260. With the present version of *DrawAid* it is not possible to nest these nuts as a higher level group, as only one level of nesting is allowed. The programmer should use grouping judiciously to minimise the number of objects at the top level of the *Draw* file.

```
 10 REM  >TutorialD
 20 LIBRARY "<DrawAid_Lib$Dir>.Procedures"
 30 PROC_report_errors
 40 PROC_DrawAid("tutorial_d")
 50 PROC_finish
 60 END
 70
 80 DEFPROC_Objects
 90 REM Draw bolted flange with nuts
100 bolt_size=10   :REM bolt 10mm diameter
110 radius=bolt_size/2
120 pcr=50         :REM pcr=pitch circle radius
130 nob=8          :REM nob=number of bolts
140 xcentre=100:ycentre=100
150 outer_radius=pcr+bolt_size
160 inner_radius=pcr-bolt_size
170 PROC_ring(width1,black,grey1,xcentre,ycentre,inner_radius,outer_radius)
180 FOR bolt=1 TO nob
190 angle=bolt*2*PI/nob+PI/nob
200 xc=xcentre+pcr*COS(angle)
210 yc=ycentre+pcr*SIN(angle)
220 PROC_new_group
230 PROC_polygon(width1,black,dark_blue,xc,yc,1.6*radius/COS(RAD(30)),6,0)
240 PROC_circle(width1,black,light_blue,xc,yc,1.6*radius)
250 PROC_circle(width1,black,grey3,xc,yc,radius)
260 PROC_end_group
270 NEXT bolt
280 ENDPROC
```

Of course further grouping is always possible by editing the file within *Draw* itself.

Instead of using two circles to produce the grey ring **PROC_ring( )** is used at line 170. This gives a transparent centre, instead of a white disc as in *TutorialC*. This means that any object behind the flange will be partly visible through the hole. See *Drawing2* of the welcome demonstration. Only three **Standard Objects**, have been used in these tutorial examples so far, but the method of use is exactly the same for the other objects which are defined in Section **4 : Standard Objects**. The parameters used by each procedure can be defined either as constants, or variables whose value is constantly being recalculated inside any of the structures of BASIC such as REPEAT...UNTIL, or WHILE.... ENDWHILE.



f) A Few Words of Text

The use of the three types of text object are demonstrated in *TutorialE*. A centre for the drawing is defined in line 100, then the text string **"System Font"** is placed, in red letters, 15mm to the left of this by line 110. The words **"Vector Text"** are then disposed on a circular arc by the loop 130 to 180. The 10 mm high characters are drawn individually, in dark blue, bold, regular font by **PROC_vector_text( )** at line 170. Each character is read in turn from the data at line 190. Note that, as two calls are made to **PROC_Objects** by **PROC_DrawAid( )** then the data pointer needs to be re-set at line 90, by use of the RESTORE command. If there is more than one character in the text string then these are grouped automatically. Enter this

program and run it as described above. Load the resulting *Draw* file into *Draw* and after selecting the main object use the **ungroup** command. You will find that each character of the string **"Vector Text"** is an individual object, whereas all the characters of **"Fancy Vector"** are grouped together in each colour. The lines 210 to 240 produce these six superimposed strings of **Vector** text giving a grey shadow effect. Lines 250 and 260 print the foreground text in yellow with a red outline. By varying the weight and angle of the characters interesting effects can be produced.

```
 10 REM  >TutorialE
 20 LIBRARY "<DrawAid_Lib$Dir>.Procedures"
 30 PROC_report_errors
 40 PROC_DrawAid("tutorial_e")
 50 PROC_finish
 60 END
 70
 80 DEFPROC_Objects
 90 RESTORE
100 xc=100:yc=100
110 PROC_text(red,xc-15,yc,"System Font")
120 radius=30
130 FOR angle = 180 TO 0 STEP -17
140 xo=xc+radius*COS(RAD(angle))
150 yo=yc+radius*SIN(RAD(angle))
160 READ char$
170 PROC_vector_text(dark_blue,xo,yo,10,bold,regular,angle-90,char$)
180 NEXT
190 DATA "V","E","C","T","O","R"," ","T","E","X","T"
200
210 FOR i%=1 TO 6
220 weight=0.35-0.05*i%:colour=i%
230 PROC_vector_text(colour,55,70,13,weight,20,4,"Fancy Vector")
240 NEXT i%
250 PROC_vector_text(red,55,70,12,bold,regular,0,"Fancy Vector")
260 PROC_vector_text(yellow,55,70,12,light,regular,0,"Fancy Vector")
270
280 PROC_outline_text("Trinity.Medium",black,white,50,150,10,25,"Tall Acorn Trinity.Medium",0)
290
300 ENDPROC
```

An example of using the Acorn outline font is given at line 280. The last parameter is the text angle which is set to 0. Try changing this to say 5 (degrees).

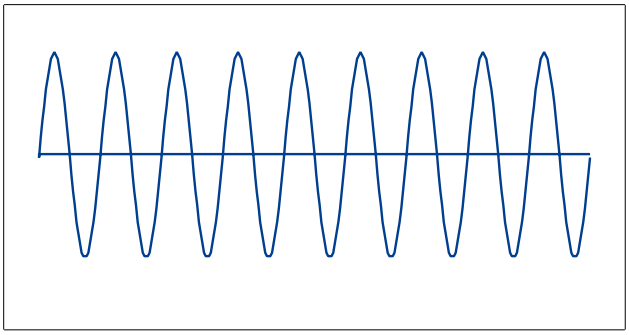
g) A Path to Progress

The *DrawAid* Standard Objects which have been used in the above tutorial examples become inefficient for more complex objects, such as mathematically defined curves. If such a curve was constructed from many line objects it would store much redundant data. Consequently a set of procedures to construct Path Objects are given in Section **6: Path Objects**.

TutorialF shows how to generate nine cycles of a SIN wave as a single path object. The path is initiated at line 90 to be a blue line, with no fill colour. Line 100 defines the location of the start of the line, and the first segment to be drawn is the horizontal axis in line 110. Individual points on the SIN wave are then calculated by the FOR...NEXT loop from 120 to 150. Because the screen scale is in mm a suitable amplitude of 30mm has been set in line 120, and the x scale, which is in degrees, has been converted to a suitable length by dividing by 20 in line 140. As the step size is set at 10 degrees, the path consists of some 325 elements. Note that there is a limit of 1000 points on a path. The path must be ended, and this is done by the procedure call at line 160.

If this program is run, and the resulting *Draw* file *Tutorial_f* is inspected within *Draw* using <Select>Edit> from the main menu, then all of the line elements of the path will be revealed. (Try entering that one by mouse!)

```
 10 REM  >TutorialF
 20 LIBRARY "<DrawAid_Lib$Dir>.Procedures"
 30 PROC_report_errors
 40 PROC_DrawAid("tutorial_f")
 50 PROC_finish
 60 END
 70
 80 DEFPROC_Objects
 90 PROC_new_path(width2,dark_blue,none)
100 PROC_move(180,100)
110 PROC_draw(18,100)
120 FOR x%=360 TO 3600 STEP 10
130 y%=100+30*SIN(RAD(x%))
140 PROC_draw(x%/20,y%)
150 NEXT x%
160 PROC_end_path
170 ENDPROC
```

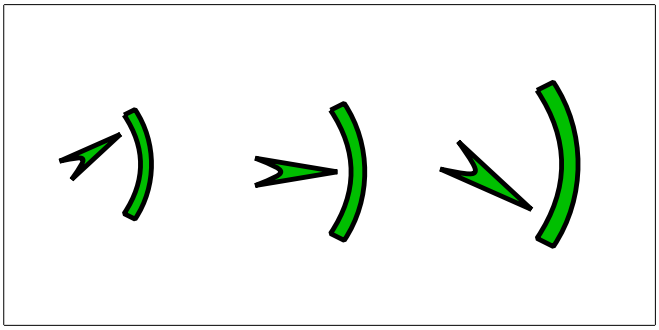


tutorial_f

Complex paths can be produced in this way. For instance the flange with holes in *TutorialC* above could be described as one continuous path containing both **PROC_draw( )** and **PROC_ move( )** calls. In this way truly hollow rings, with "real" bolt holes, can be defined. The path sequence can be the subject of the user's own procedure definition. The resulting Special Object, say **PROC_flange( )**, can then be called, using the user's own dimension values each time as required.

h) Pointing the Way

As a further example of such a user defined Special Object let us suppose that a dial object with a positionable pointer is required. The program *TutorialG* gives an example of how this might be done, and the figure below shows three pointers.



tutorial_g

These pointers are produced in *TutorialG* by the user defined procedure **PROC_meter(x,y,reading%, size)**. Lines 90 to 110, within **DEFPROC_Objects**, make three calls to this procedure, producing three sizes of the pointer at three locations, and at three different indications, +100%, 0, and -100%. Any intermediate position could equally well be produced.

```
 10 REM  >TutorialG
 20 LIBRARY "<DrawAid_Lib$Dir>.Procedures"
 30 PROC_report_errors
 40 PROC_DrawAid("tutorial_g")
 50 PROC_finish
 60 END
 70
 80 DEFPROC_Objects
 90 PROC_meter(100,100,100,0.8)
100 PROC_meter(130,100,0,1.0)
110 PROC_meter(160,100,-100,1.2)
120 ENDPROC
130
140 DEFPROC_meter(x,y,reading%,size)
150 IF reading% >100 THEN reading% = 100
160 IF reading% <-100 THEN reading% = -100
170 PROC_new_path(width2,black,light_green)
180 PROC_move(10,0)
190 PROC_draw(-2,2)
200 PROC_curve(-2,-2,3,0,3,0)
210 PROC_draw(10,0)
220 PROC_rotate_path(0,0,reading%/3)
230 PROC_move(9,9)
240 PROC_curve(9,-9,13,3,13,-3)
250 PROC_draw(11,-10)
260 PROC_curve(11,10,15,-4,15,4)
270 PROC_draw(9,9)
280 PROC_scale_path(size)
290 PROC_locate_path(0,0,x,y)
300 PROC_end_path
310 ENDPROC
```

Within **DEFPROC_meter( )** the pointer is initially designed in the centre or zero position by the path starting at its tip. Line 180 defines this to be at the arbitrary location (10,0). Since the path is finally located by line 290, this origin point is selected for arithmetical convenience. Lines 190 to 210 define the pointer, which is then rotated about location (0,0) to the required angle by line 220. Note that the path does not need to be completed at this point, and rotation applies only to those points defined prior to this instruction. The path continues with a move to the top end of the scale at line 230, and the scale is defined by the next four statements. The whole path

is then scaled to size at line 280. Note that, like **PROC_rotate( )**, this scaling could be applied to only the first part of the path if required. The path is then located at the point **x,y** passed as **PROC_meter( )** parameters. Finally the path definition ends at line 300.

i) Multiple File Output

In the above example it may be desired to save each pointer as a separate file. The necessary changes are given in the listing *TutorialH*.

```
 10 REM  >TutorialH
 20 LIBRARY "<DrawAid_Lib$Dir>.Procedures"
 30 LIBRARY "<DrawAid_Lib$Dir>.Instrument"
 40 PROC_report_errors
 50 FOR meter%=1 TO 3
 60 filename$="Meter"+STR$(meter%)
 70 PROC_DrawAid(filename$)
 80 NEXT meter%
 90 PROC_finish
100 END
110
120 DEFPROC_Objects
130 IF meter%=1 THEN PROC_meter(10,10,100,0.8)
140 IF meter%=2 THEN PROC_meter(10,10,0,1.0)
150 IF meter%=3 THEN PROC_meter(10,10,-100,1.2)
160 ENDPROC
```

The *Draw* files are generated within the loop 40 to 70, with the filename varying for each call to **PROC_DrawAid( )**. The drawings are stored individually as *Meter1*, *Meter2*, and *Meter3*. The procedure **PROC_meter( )** having been proved has now been added to a user Library named *Instrument*.

Gears

This facility enables a user to produce whole ranges of parametric drawings, of similar objects which vary with the value of any number of parameters. Examples could be gear wheels, multi-pin chip symbols, poly-molecule diagrams, or a range of window modular frames.

i) Further Exercises.

Many further examples are included in the *Examples* directory, and in the main text of the Guide.

: Notes

19: Tools

1) MessageMon

If the BASIC program *MessageMon* is dragged onto the installed *DrawAid* icon then a 32kByte task window is opened. This *Edit* task will record any input and output from following user's program together together with *DrawAid* status messages. If the <Shift> key is held down whilst clicking on this task window Close icon, then the window will be "iconised" onto the background. However, the task is still running, and can be opened up later by clicking on the icon. This is a useful tool for debugging as it can be used to trace execution of the program.

2) Palettes

For use with multiple grey levels the palettes *Grey* and *Default* are included.

3) BasicEd

As Version 3 of *DrawAid* can only be run on RISC OS 3.1 all users will have *Edit*, consequently the tool *BasicEd* is no longer included here.

4) BasicTask

The functions of the application *BasicTask* which was supplied with previous versions of *DrawAid*, are now incorporated into the *DrawAid* application.

: Notes

20 : Upgrade Notes

DrawAid Version 3 works in essentially the same manner as previous versions. The most obvious difference is the presence of the new monitoring program, but some procedures have been extended with extra parameters. However, there should be little difficulty in adapting earlier programs to work with the new version.

A number of users of *DrawAid 2* requested that the utility have an icon on the icon bar, and behave in a fully RISC OS compliant manner. Because of problems associated with adding libraries and BASIC user programs to a Wimp program a separate application now provides an interface with the user. This independent program has assumed the *DrawAid* name and icon, and interacts with the user's program through the facility of system variables to exchange messages. It has also taken on the task of illustrating the files produced, using the Acorn *DrawFile* module.

Changes which will be required to existing programs are to the procedures:

PROC_sprite( ), and PROC_outline_text( )

Both of these procedures now have an extra parameter (the last in each parameter list) which defines the rotation angle. If this parameter is not added to earlier programs. the user will receive an "Arguments of function/procedure incorrect :" message. However, old programs are easy to change with careful use of the Global change facilities available on most editors.

The procedures **PROC_dimensions()** and **PROC_dimensions_off(),** which previously used the supplied Vector font, now use the Acorn outline fonts. The vector font can still be used for pen plotters by using two new associated procedures. See sections **4: Standard Objects** and **13: Other Procedures** for information about these changes.

A problem in *DrawAid2* when calculating the drawing bounding box with non **mm** units has required the introduction of the procedure **PROC_set_units()**. The reserved variable **units** is no longer used. See sections 1**2: and 14** for further details.

If you find any problems with using either old or new programs please let us know.

: Notes