OPL

GRAPHICS & FRIENDLIER INTERACTION

This part of the OPL User Guide is divided into two sections:

- **Graphics: this covers the powerful graphics capabilities of OPL.**
- **Friendlier Interaction: this explains how to make your programs easier to use through employing features such as menus and dialogs.**

CONTENTS

GRAPHICS

OPL graphics allows you, for example, to:

- **Draw lines and boxes.**
- **Fill areas with patterns.**
- **Display text in a variety of styles, at any position on the screen.**
- **Scroll areas of the screen.**
- **Manipulate windows and bit patterns.**
- **Read data back from the screen.**

On the Series 5 you can additionally:

- **Draw circles and ellipses.**
- **Set the pen width.**

You can draw using black, grey and white.

Graphics keywords begin a `G`. In the OPL User Guide a lower case `g` is used - for example, `gBOX` - but you can type them using upper or lower case letters.

⚠ **Some graphics keywords are mentioned only briefly in this section. For more details about them, see the 'Alphabetic Listing' section of the 'Glossary.pdf' document.**

SIMPLE GRAPHICS

The Psion screens are made up of the following numbers of points:

Series 5	*Series 3c*	*Siena*
640 × 240	480 × 160	240 × 160

These points are sometimes referred to as *pixels*.

Each pixel is identified by two numbers, giving its position across and down from the top left corner of the screen. 0,0 denotes the pixel in the top left corner; 2,1 is the pixel 2 points across and one down, and so on. 639,239 is the pixel in the bottom right corner on the Series 5, for example.

Note that these co-ordinates are very different to the cursor positions set by the AT command.

OPL maintains a *current position* on the screen. Graphics commands which draw on the screen generally take effect at this position. Initially, the current position is the top left corner, 0,0.

❺ On the Series 5, colour modes allowing the use of 2, 4 or 16 colours are available. These colours are mapped to grey shades on a non-colour screen.

❸ On the Series 3c and Siena, you can draw using black, grey and white although grey is not accessible by default (it has to be switched on explicitly). See the 'Drawing in grey' section below for further details.

DRAWING LINES

Here is a simple procedure to draw a horizontal line in the middle of the screen:

```
PROC lines:
    gMOVE 180,80
    gLINEBY 120,0
    GET
ENDP
```

gMOVE moves the current position by the specified amount. In this case, it moves the current position 180 pixels right and 80 down, from 0,0 to 180,80. It does not display anything on the screen.

gLINEBY (g-line-by) draws a line from the current position (just set to 180,80) to a point at the distance you specify - in this case 120 to the right and 0 down, i.e. 300,80.

❺ The Series 5 never draws the end point of lines: for `gLINEBY dx%,dy%`, point `gX+dx%,gY+dy%` is not drawn. Note, however, that OPL specially plots the point when the start and end-point coincide.

❸ When drawing a horizontal line, as in the above example, the line that is drawn **includes** the pixel with the lower x co-ordinate and **excludes** the pixel with the higher x co-ordinate. Similarly, when drawing a vertical line, the line **includes** the pixel with the lower y co-ordinate and **excludes** the pixel with the higher y co-ordinate.

When drawing a diagonal line, the co-ordinates of the end pixels are turned into a rectangle. The top left pixel lies inside the boundary of this rectangle and the bottom right pixel lies outside it. The line drawing algorithm then fills in those pixels that are intersected by a mathematical line between the corners of the rectangle. Thus the line will be drawn minus one or both end points.

gLINEBY also has the effect of moving the current position to the end of the line it draws.

With both gMOVE and gLINEBY, you specify positions **relative to the current position**. Most OPL graphics commands do likewise. gMOVE and gLINEBY, however, do have corresponding commands which use absolute pixel positions. gAT moves to the pixel position you specify; gLINETO draws a line from the current position to an absolute position. The horizontal line procedure could instead be written:

```
PROC lines:
    gAT 180,80
    gLINETO 300,80
    GET
ENDP
```

gAT and gLINETO may be useful in very short graphics programs, and gAT is always the obvious command for moving to a particular point on the screen, before you start drawing. But once you do start drawing, use gMOVE and gLINEBY. They make it much easier to develop and change programs, and allow you to make useful graphics procedures which can display things anywhere you set the current position. Almost all graphics drawing commands use relative positioning for these reasons.

DRAWING DOTS

You can set the pixel at the current position with `gLINEBY 0,0`.

RIGHT AND DOWN, LEFT AND UP

gMOVE and gLINEBY find the position to use by adding the numbers you specify on to the current position. If the numbers are positive, it moves to the right and down the screen. If you use negative numbers, however, you can specify positions to the left of and/or above the current position. For example, this procedure draws the same horizontal line as before, then another one above it:

```
PROC lines2:
    gMOVE 180,80
    gLINEBY 120,0
    gMOVE 0,-20
    gLINEBY -120,0
    GET
ENDP
```

The first two program lines are the same as before. gLINEBY moves the current position to the end of the line it draws, so after the first gLINEBY the current position is 300,80. The second gMOVE moves the current position **up** by 20 pixels; the second gLINEBY draws a line to a point 120 pixels to the **left**.

❺ The end pixel is never set;

❸ For horizontal and vertical lines, the right-hand/bottom pixel is not set. For diagonal lines, the right-most and bottom-most pixels are not set; these may be the same pixel.

GOING OFF THE SCREEN

No error is reported if you try to draw off the edge of the screen. It is quite possible to leave the current position off the screen - for example, `gLINETO 650,80` will draw a line from the current position to some point on the right-hand screen edge, but the current position will finish as 650,80.

There's no harm in the current position being off the screen. It allows you to write procedures to display a certain pattern at the current position, and not have to worry whether that position is too close to the screen edge for the whole pattern to fit on.

CLEARING THE SCREEN

`gCLS` clears the screen.

DRAWING IN GREYS ON THE SERIES 5

COLOUR MODES

On the Series 5, OPL supports various *colour modes*:

- 2-colour mode (black and white). This is stored as 1 bit per pixel (bpp).
- 4-colour mode (white, light grey, dark grey and black). This is stored as 2 bpp.
- 16-colour mode (white, 14 greys and black). This is stored as 4 bpp.

By default the screen is in 4-colour mode, so black, white and two greys are automatically available. To enable drawing in 16 colours, you need to use `DEFAULTWIN 2` at the start of your program. (Note that this also clears the screen.) 16-colour mode is not automatically available because the hardware switches to 16-colour mode if any window displayed has this mode and 16-colour mode uses twice the memory and much more power than 4-colour mode. So the default ensures that programs that do not need to use 16 colours are not unnecessarily penalised.

The display power consumption is dependent on both the colour mode and the pattern on the display, as follows:

- Colour mode: power consumption doubles from 1 bpp to 2 bpp and again from 2 bpp to 4 bpp.
- Pattern: the worst power consumption for the display is produced by a checker board pattern.

Grey areas also increase the power consumption considerably.

Overall the current taken by the display is between 25% and 60% of the total idle current:

- 25%: 2 bpp plain screen (e.g. plain Word screen).
- 40%: 2 bpp grey areas on screen (e.g. Calc screen).
- 60%: 4 bpp bitmap on the screen.

It is difficult to be more precise since the power consumption is very dependent on what is being displayed.

To set the colour used for graphics, use `gCOLOR red%,green%,blue%`. The `red%`, `green%` and `blue%` values specify a colour, which will be mapped to white, black or one of the greys on non-colour screens. Note that if the values of `red%`, `green%` and `blue%` are equal, then a pure grey results in a 16-colour window, ranging from black (0) to white (255).

Note that if you use gCOLOR in 4-colour mode, colours with shades between the four colours available will appear *dithered*, that is areas of the colour will have some pixels set to one colour and some to another so as to give the appearance of a colour between the two colours used. If gCOLOR is used in 2-colour mode, light greys will be mapped to white and dark greys to black.

`DEFAULTWIN 1` disables the use of 16 colours again, returning to 4-colour mode and also clearing the screen. If you do not wish to use any greys then you should use `DEFAULTWIN 0` to use 2-colour mode. N.B. This is in fact implemented by mapping dark grey to black and light grey to white.

`DEFAULTWIN` does not effect PRINT statements - it applies only to graphics and graphics text (see gPRINT later).

Constants for the modes of DEFAULTWIN are supplied in Const.oph. See the 'Calling Procedures' section of the 'Basics.pdf' document for details of how to use this file and see Appendix E in the 'Appends.pdf' document for a listing of it.

NO CONCEPT OF A GREY PLANE: GREY IS JUST ONE OF THE COLOURS

There is no concept of a grey plane on the Series 5 (see the Series 3c description below): `gGREY mode%` just draws in grey (unless `mode%=0`, when it draws in black).

DRAWING IN GREY ON THE SERIES 3C AND SIENA

INITIALISING FOR THE USE OF GREY

To draw in grey you need to use `DEFAULTWIN 1` at the start of your program. (Note that this clears the screen.) Grey is not automatically available because it requires twice the memory (and takes longer to scroll or move) compared to having just black. So programs that do not need to use grey are not unnecessarily penalised.

`DEFAULTWIN 0` disables the use of grey again, also clearing the screen.

It is not possible to have a screen using grey only.

`DEFAULTWIN 1` does not cause PRINT to print in grey - it applies only to graphics and graphics text (see gPRINT later).

When you use `DEFAULTWIN 1` the existing black-only screen is cleared and replaced by one which contains a *black plane* **and also** a *grey plane*. The black plane is also sometimes called the normal plane. These are referred to as 'planes' because intuitively it is simplest to think of there being a plane of black pixels **in front of** (or on top of) a plane of grey pixels, with any grey only ever visible if the black pixel in front of it is clear.

If you draw a pixel using both black and grey, it will appear black. If you then clear the black plane only, the same pixel will appear grey. If you draw a pixel using grey only it will appear grey unless it is already black, in which case it is effectively hidden behind the black plane.

If you need to use grey, you are recommended to use `DEFAULTWIN 1` once and for all at the start of your program. One reason is because DEFAULTWIN can fail with a 'No system memory' error and it is unlikely that you would want to continue without grey after trying to enable it.

Note that gXBORDER, gBUTTON and gDRAWOBJECT all use grey and therefore can only be used when grey in enabled. If grey is not enabled, they raise a 'General failure' error.

USING GREY

Once you have used `DEFAULTWIN 1` you can use the gGREY command to set which plane should be used for all subsequent graphics drawing (until the next use of gGREY).

`gGREY 0`	draws to the black plane only.
`gGREY 1`	draws to the grey plane only.
`gGREY 2`	draws to both planes.

`gGREY 1` and `gGREY 2` raise an error if the current window does not have a grey plane.

As mentioned earlier, when you set a pixel using both black and grey, the pixel appears black because the black plane is effectively in front of the grey plane. So drawing to both planes is generally only used for clearing pixels. For example, if your screen has both black and grey pixels, gCLS will clear the pixels only in the plane selected by gGREY. To clear the whole screen with gCLS, you therefore need `gGREY 2`.

To draw in grey when the pixels to which you are drawing are currently black, you first need to clear the black.

A pixel will appear white only if it is clear in both planes.

EXAMPLE

The following procedure initialises the screen to allow grey, draws a horizontal line in grey, another below it in black only and a third below it in both black and grey. Pressing a key clears the black plane only, revealing the grey behind the black in the bottom line and clearing the middle line altogether.

```
PROC exgrey:
    DEFAULTWIN 1                                    REM enable grey
    gAT 0,40 :gGREY 1 :gLINEBY 480,0                REM grey only
    gAT 0,41 :gLINEBY 480,0
    gAT 0,80 :gGREY 0 :gLINEBY 480,0                REM black only
    gAT 0,81 :gLINEBY 480,0
    gAT 0,120 :gGREY 2 :gLINEBY 480,0               REM both planes
    gAT 0,121 :gLINEBY 480,0
    GET
    gGREY 0                                         REM black only
    gCLS                                            REM clear it
    GET
ENDP
```

OVERWRITING PIXELS

DRAWING RECTANGLES

The gBOX command draws a box outline. For example, `gBOX 100,20` draws a box from the current position to a point 100 pixels to the right and 20 down. If the current position were 200,40, the four corners of this box would be at 200,40, 300,40, 300,60 and 200,60.

❺ If you have used gCOLOR as described earlier, the box is drawn in the colour selected.

❸ If you have used `DEFAULTWIN 1` and gGREY as described earlier, the box is drawn to the black and/or grey plane as selected.

gBOX does not change the current position.

gFILL draws a filled box in the same way as gBOX draws a box outline, but it has a third argument to say which pixels to set. If set to 0, the pixels which make up the box would be set. If set to 1, pixels are cleared; if set to 2, they are inverted, that is, pixels already set on the screen become cleared, and vice versa. The values 1 and 2 are used when **overwriting** areas of the screen which already have pixels set.

❸ If you have used `DEFAULTWIN 1` and gGREY as described earlier, the filled box will be set, cleared or inverted in the black and/or grey plane as selected. Once again, it helps to think of the pixels being set or clear in each plane independently: so clearing the pixel in the black plane reveals the grey plane behind it where the pixel may be set or clear.

So with `gGREY 1` set for drawing to the grey plane only, inverting the pixels in the filled box will change the grey plane only - black pixels are left alone but clear or grey pixels are inverted to grey and clear pixels respectively. Similarly, inverting the black plane changes clear pixels to black, but “clearing” black pixels displays grey if the pixel is set in the grey plane.

This procedure displays a “robot” face, using gFILL to draw set and cleared boxes:

```
PROC face:
    gFILL 120,120,0                  REM set the entire face
    gMOVE 10,20 :gFILL 30,20,1       REM left eye
    gMOVE 70,0 :gFILL 30,20,1        REM right eye
    gMOVE -30,30 :gFILL 20,30,1      REM nose
    gMOVE -20,40 :gFILL 60,20,1      REM mouth
    GET
ENDP
```

Before calling such a procedure, you would set the current position to be where you wanted the top left corner of the head.

You could make the robot wink with the following procedure, which inverts part of one eye:

```
PROC wink:
    gMOVE 10,20                      REM move to left eye
    gFILL 30,14,2                    REM invert most of the eye
    PAUSE 10
    gFILL 30,14,2                    REM invert it back again
    GET
ENDP
```

Again, you would set the current position before calling this.

The gPATT command can be used to draw a shaded filled rectangle. To do this, use `-1` as its first argument, then the same three arguments as for gFILL - width, height, and overwrite method. Overwrite methods 0, 1 and 2 apply only to the pixels which are ‘on’ in the shading pattern. Whatever was on the screen may still show through, as those pixels which are ‘clear’ in the shading pattern are left as they were.

To completely overwrite what was on the screen with the shaded pattern, gPATT has an extra overwrite method of 3. So, for example, `gPATT -1,120,120,3` in the first procedure would have displayed a shaded robot head, whatever may have been on the screen.

❸ Again, the shaded pattern will be drawn in grey if you have selected the grey plane only using `gGREY 1`. And again, if you are writing to the black plane only, any pixels set in the grey plane can be seen if the corresponding pixels in the black plane are clear.

OVERWRITING WITH ANY DRAWING COMMAND

By using the gGMODE command, any drawing command such as gLINEBY or gBOX can be made to clear or invert pixels, instead of setting them. gGMODE determines the effect of **all** subsequent drawing commands.

The values are the same as for gFILL: `gGMODE 1` for clearing pixels, `gGMODE 2` for inverting pixels, and `gGMODE 0` for setting pixels again. (0 is the initial setting.)

For example, some white lines can give the robot a furrowed brow:

```
PROC brow:
    gGMODE 1                    REM gLINEBY will now clear pixels
    gMOVE 10,8 :gLINEBY 100,0
    gMOVE 0,4 :gLINEBY -100,0
    gGMODE 0
    GET
ENDP
```

❸ The setting for gGMODE applies to the planes selected by gGREY. With `gGREY 1` for instance, `gGMODE 1` would cause gLINEBY to clear pixels in the grey plane and `gGMODE 0` to set pixels in the grey plane.

❺ Constants for the modes are supplied in Const.oph. See the 'Calling Procedures' section of the 'Basics.pdf' document for details of how to use this file and see Appendix E in the 'Appends.pdf' document for a listing of it.

OTHER DRAWING KEYWORDS

For more details of these keywords, see the 'Alphabetic Listing' section of the 'Glossary.pdf' document.

- gBUTTON: draw a 3-D button (a picture of a key, not of an application button) enclosing supplied text. The button can be raised, depressed or sunken. On the Series 5, the button may also enclose a *bitmap* with a *mask* (see the 'Bitmaps' section below).
- gBORDER, gXBORDER: draw 2-D/3-D borders.
- gINVERT: invert a rectangular area, except for its four corner pixels.
- gCOPY: copy a rectangular area from one position on the screen to another. On the Series 3c, both black and grey planes are copied.
- gSCROLL: move a rectangular area from one position on the screen to another, or scroll the contents of the screen in any direction. On the Series 3c, both black and grey planes are moved.
- gPOLY: draw a sequence of lines.
- ❺ gCIRCLE, gELLIPSE: draw a cirlce or ellipse which can be filled or empty.
- ❺ gSETPENWIDTH: draw with a different pen width.
- ❸ gDRAWOBJECT: draw a *graphics object*. This can be used to draw the "lozenge" used to display the words 'City' and 'Home' in the World application.
- ❸ Note that commands such as gSCROLL, which move existing pixels, affect both black and grey planes. gGREY only restricts drawing and clearing of pixels.

GRAPHICAL TEXT

DISPLAYING TEXT WITH GPRINT

The PRINT command displays text in one font, in a screen area controlled by the FONT or SCREEN commands. You can, however, display text in a variety of fonts and styles, at any pixel position, with gPRINT.

❺ gPRINT also allows you to draw text in grey if you have used the gCOLOR command previously.

❸ gPRINT also lets you draw text to the grey plane, if you have used DEFAULTWIN and gGREY (discussed earlier).

You can (to a lesser degree on the Series 3c) control the font and style used by OPL's other text-drawing keywords, such as PRINT and EDIT. 'The text and graphics windows' at the end of this section.

gPRINT is a graphical version of PRINT, and displays a list of expressions in a similar way. Some examples:

```
gPRINT "Hello",name$
gPRINT a$
gPRINT "Sin(PI/3) is",sin(pi/3)
```

Unlike PRINT, gPRINT does not end by moving to a new line. A comma between expressions is still displayed as a space, but a semicolon has no effect. `gPRINT` used on its own does nothing.

The first character displayed has its left side and baseline at the current position. The baseline is like a line on lined note paper graphically, this is the horizontal line which includes the lowest pixels of upper case characters. Some characters, such as 'g', 'j', 'p', 'q' and 'y', set pixels below the baseline.

After using gPRINT, the current position is at the end of the text so that you can print something else immediately beyond it. As with other graphics keywords, no error is reported if you try to display text off the edge of the screen.

While `CURSOR ON` displays a flashing cursor for ordinary text displayed with PRINT, `CURSOR 1` switches on a cursor for graphical text which is displayed at the current position. `CURSOR OFF` removes either cursor.

FONTS

The gFONT command sets the font to be used by subsequent gPRINT commands.

A large set of fonts which can be used with gFONT is provided in the Psion's ROM. In the following list, Swiss and Arial fonts refer to fonts without serifs while Roman and Times fonts either have serifs (e.g. font 6) or are in a style designed for serifs but are too small to show them (e.g. font 5 on the Series 3c). Courier is a *mono-spaced* font. *Mono-spaced* fonts have characters which all have the same width (and have their 'pixel size' listed as width `x` height); in *proportional* fonts each character can have a different width.

Fonts 1,2 and 3 are the Series 3 fonts, used when running in compatibility mode. Therefore these fonts are not supported on the Series 5.

Font number	*Series 5 font name*	*height in pixels*	*Series 3c font name*	*size in pixels*
1	-	-	Series 3 normal	8
2	-	-	Series 3 bold	8
3	-	-	Series 3 digits	6x6
4	Courier	8	Mono	8x8
5	Times	8	Roman	8
6	Times	11	Roman	11
7	Times	13	Roman	13
8	Times	15	Roman	16
9	Arial	8	Swiss	8
10	Arial	11	Swiss	11
11	Arial	13	Swiss	13
12	Arial	15	Swiss	16
13	Tiny (mono)	4	Mono	6x6

The special font number `&9a` (`$9a` on the Series 3c) is set aside to give a machine's default graphics font; this is the font used initially for graphics text. The default font is 12 (Arial 15) for the Series 5 and 11 (Swiss 13) for the Series 3c. So `gFONT 12` (`11`) or `gFONT &9a` (`$9a` on the Series 3c) both set the standard font, which gPRINT normally uses.

❺ On the Series 5, fonts are identified by a 32-bit UID, rather than a 16-bit value representing the font position in the ROM as on the Series 3c. Series 5 OPL does however provide a mapping where possible between Series 3c OPL font IDs and Series 5 OPL font UIDs, but there are inevitably some incompatibilities.

Therefore gFONT takes a long integer argument rather than an integer. As well as being able to use the font IDs 4 to 13 (which are automatically converted to long integers for all keywords taking font IDs), you can also directly specify the fonts by UID by using the definitions listed in the header file Const.oph. (See the 'Calling Procedures' section of the 'Basics.pdf' document for details of how to use this file and see Appendix E in the 'Appends.pdf' document for a listing of it.) A wider range of fonts is also available, as you will see from Const.oph. These are Arial and Times proportional fonts and Courier mono-spaced font, each with sizes 8, 11, 13, 15, 18, 22, 27 and 32, Squashed font (which are used for the toolbar in bold style) and Tiny fonts.

For example, normal Arial proportional font with height 8 pixels has UID given by,

```
CONST KFontArialNormal8&=268435954
```

so you could use,

```
INCLUDE "Const.oph"
...
gFONT KFontArialNormal8&
```

See gINFO32 (for the Series 5) or gINFO (for the Series 3c) in the 'Alphabetic Listing' section of the 'Glossary.pdf' document if you need to find out more information about fonts.

The following programs shows you examples of the fonts. (`!!!` is displayed to emphasise the mono-spaced fonts):

❺ This program displays a variety of fonts, using both the OPL codes for them and their UIDS (the constants are defined in Const.oph - see above). The two screens should be identical.

```
INCLUDE "Const.oph"

PROC fonts:
    showfont:(4,15,"Courier 8")
    showfont:(5,25,"Times 8")
    showfont:(6,38,"Times 11")
    showfont:(7,53,"Times 13")
    showfont:(8,71,"Times 15")
    showfont:(9,81,"Arial 8")
    showfont:(10,94,"Arial 11")
    showfont:(11,109,"Arial 13")
    showfont:(12,127,"Arial 15")
    showfont:(13,135,"Tiny 44")
    GET :GCLS
    showfontbyuid:(KFontCourierNormal8&,15,"Courier 8")
    showfontbyuid:(KFontTimesNormal8&,25,"Times 8")
    showfontbyuid:(KFontTimesNormal11&,38,"Times 11")
```

```
    showfontbyuid:(KFontTimesNormal13&,53,“Times 13”)
    showfontbyuid:(KFontTimesNormal15&,71,“Times 15”)
    showfontbyuid:(KFontArialNormal8&,81,“Arial 8”)
    showfontbyuid:(KFontArialNormal11&,94,“Arial 11”)
    showfontbyuid:(KFontArialNormal13&,109,“Arial 13”)
    showfontbyuid:(KFontArialNormal15&,127,“Arial 15”)
    showfontbyuid:(KFontTiny4&,135,“Tiny 4”)
ENDP

PROC showfont:(font%,y%,str$)
    gFONT font%
    gAT 20,y% :gPRINT font%
    gAT 50,y% :gPRINT str$
    gAT 150,y% :gPRINT “!!!”
ENDP

PROC showfontbyuid:(font&,y%,str$)
    gFONT font&
    gAT 20,y% :gPRINT font%
    gAT 50,y% :gPRINT str$
    gAT 150,y% :gPRINT “!!!”
ENDP
```

❸

```
PROC fonts:
 showfont:(4,15,“Mono 8x8”)
 showfont:(5,25,“Roman 8”)
 showfont:(6,38,“Roman 11”)
 showfont:(7,53,“Roman 13”)
 showfont:(8,71,“Roman 16”)
 showfont:(9,81,“Swiss 8”)
 showfont:(10,94,“Swiss 11”)
 showfont:(11,109,“Swiss 13”)
 showfont:(12,127,“Swiss 16”)
 showfont:(13,135,“Mono 6x6”)
 GET
ENDP

PROC showfont:(font%,y%,str$)
 gFONT font%
 gAT 20,y% :gPRINT font%
 gAT 50,y% :gPRINT str$
 gAT 150,y% :gPRINT “!!!”
ENDP
```

TEXT STYLE

The gSTYLE command sets the text style to be used by subsequent gPRINT commands.

Choose from these styles:

gSTYLE 1	bold
gSTYLE 2	underlined
gSTYLE 4	inverse
gSTYLE 8	double height
gSTYLE 16	mono
gSTYLE 32	italic

❺ Constants for the styles are supplied in Const.oph. See the ‘Calling Procedures’ section of the ‘Basics.pdf’ document for details of how to use this file and see Appendix E in the ‘Appends.pdf’ document for a listing of it.

The ‘mono’ style is not proportionally spaced - each character is displayed with the same width, in the same way that PRINT displays characters (by default). A proportional font can be displayed as a mono-spaced font by setting the ‘mono’ style. See the previous section for the list of mono-spaced and proportional fonts.


It is inefficient to use the ‘mono’ style to display a font which is already mono-spaced.

You can combine these styles by adding the relevant numbers together. gSTYLE 12 sets the text style to inverse and double-height (4+8=12). Here’s an example of this style:

```
PROC style:
    gAT 20,50 :gFONT 11
    gSTYLE 12 :gPRINT “Attention!”
    GET
ENDP
```

Use gSTYLE 0 to reset to normal style.

The bold style provides a way to make any font appear bold. Except for the smaller fonts on the Series 3c, most Psion fonts look reasonably bold already. Note that using the bold style sometimes causes a change of font; if you use gINFO you may see the font name change.

❺ Note that fonts which are always bold are available on the Series 5. Using a bold style with these fonts results in a double bold font. It is not necessary to use these fonts to produce bold font; you can of course use just the normal fonts in a bold style.

OVERWRITING WITH GPRINT

gPRINT normally displays text as if writing it with a pen - the pixels that make up each letter are set, and that is all. If you’re using areas of the screen which already have some pixels set, or even have all the pixels set, use gTMODE to change the way gPRINT displays the text.

gTMODE controls the display of text in the same way as gGMODE controls the display of lines and boxes. The values you use with gTMODE are similar to those for gGMODE: `gTMODE 1` for clearing pixels, `gTMODE 2` for inverting pixels, and `gTMODE 0` for setting pixels again. There is also `gTMODE 3` which sets the pixels of each character while clearing the character’s background. This is very useful as it guarantees that the text is readable.

❸ As for gGMODE, the setting for gTMODE applies to the planes selected by gGREY. With `gGREY 1` for instance, `gTMODE 1` would cause gPRINT to clear pixels in the grey plane and `gTMODE 0` to set pixels in the grey plane.

❺ Constants for the modes of gTMODE are supplied in Const.oph. See the 'Calling Procedures' section of the 'Basics.pdf' document for details of how to use this file and see Appendix E in the 'Appends.pdf' document for a listing of it.

These procedures (for the Series 5 and the Series 3c respectively) shows the various effects possible via gTMODE:

❺
```
PROC tmode:
 DEFAULTWIN 2
 gFONT 11 :gSTYLE 0
 gAT 160,0 :gFILL 160,80,0         REM Black box
 gAT 220,0 :gFILL 40,80,1          REM White box
 gAT 180,20 :gTMODE 0 :gPRINT "ABCDEFGHIJK"
 gAT 180,35 :gTMODE 1 :gPRINT "ABCDEFGHIJK"
 gAT 180,50 :gTMODE 2 :gPRINT "ABCDEFGHIJK"
 gAT 180,65 :gTMODE 3 :gPRINT "ABCDEFGHIJK"
 gCOLOR $50,$50,$50
 gAT 160,80 :gFILL 160,80,0                REM Grey box
 gAT 220,80 :gFILL 40,80,1                 REM White box
 gAT 180,100 :gTMODE 0 :gPRINT "ABCDEFGHIJK"
 gAT 180,115 :gTMODE 1 :gPRINT "ABCDEFGHIJK"
 gAT 180,130 :gTMODE 2 :gPRINT "ABCDEFGHIJK"
 gAT 180,145 :gTMODE 3 :gPRINT "ABCDEFGHIJK"
 GET
ENDP
```

❸
```
PROC tmode:
 DEFAULTWIN 1                              REM enable grey
 gFONT 11 :gSTYLE 0
 gAT 160,0 :gFILL 160,80,0                 REM Black box
 gAT 220,0 :gFILL 40,80,1                  REM White box
 gAT 180,20 :gTMODE 0 :gPRINT "ABCDEFGHIJK"
 gAT 180,35 :gTMODE 1 :gPRINT "ABCDEFGHIJK"
 gAT 180,50 :gTMODE 2 :gPRINT "ABCDEFGHIJK"
 gAT 180,65 :gTMODE 3 :gPRINT "ABCDEFGHIJK"
 gGREY 1
 gAT 160,80 :gFILL 160,80,0                REM Grey box
 gAT 220,80 :gFILL 40,80,1                 REM White box
 gAT 180,100 :gTMODE 0 :gPRINT "ABCDEFGHIJK"
 gAT 180,115 :gTMODE 1 :gPRINT "ABCDEFGHIJK"
 gAT 180,130 :gTMODE 2 :gPRINT "ABCDEFGHIJK"
 gAT 180,145 :gTMODE 3 :gPRINT "ABCDEFGHIJK"
 GET
ENDP
```

OTHER GRAPHICAL TEXT KEYWORDS

- gPRINTB: display text left aligned, right aligned or centred, in a cleared box. The gTMODE setting is ignored.

  ❸ With `gGREY 1`, only grey background pixels in the box are cleared and with `gGREY 0`, only black pixels; with `gGREY 2` all background pixels in the box are cleared.

- gXPRINT: display text underlined/highlighted.
- gPRINTCLIP: display text clipped to whole characters.
- gTWIDTH: find width required by text.

All of these keywords take the current font and style into account, and work on **a single string**. On the Series 5, they display text in the colour specified by gCOLOR. On the Series 3c, they display the text in black or grey according to the current setting of gGREY.

WINDOWS

So far, you've used the whole of the screen for displaying graphics. You can, however, use *windows* - rectangular areas of the screen.

❺  Sprites (described in the 'OPX.pdf' document) can display non-rectangular shapes.

OPL allows a program to use up to sixty-four windows at any one time.

❸  Sprites (described in the 'Advanced.pdf' document) can display non-rectangular shapes.

OPL allows a program to use up to eight windows at any one time.

WINDOW IDS AND THE DEFAULT WINDOW

Each window has an ID number, allowing you to specify which window you want to work with at any time.

When a program first runs, it has one window called the *default window*. Its ID is 1, it is the full size of the screen, and initially all graphics commands operate on it. (This is why '0,0' has so far referred to the top left of the screen: it is true for the default window.)

Other windows you create will have IDs from 2 to 64 (2 to 8 on the Series 3c). **When you make another window it becomes the current window, and all subsequent graphics commands operate on it.**

The first half of this section used only the default window. However, everything actually applies to the current window. For example, if you make a small window current and try to draw a very long line, the current position moves off past the window edge, and only that part of the line which fits in the **window** is displayed.

GRAPHICS KEYWORDS AND WINDOWS

For OPL graphics keywords, **positions apply to the window you are using at any given time**. The point 0,0 means the top left corner of the current window, not the top left corner of the screen.

❺ Each window can be created in any of the 3 colour modes by specifying the last argument in the gCREATE command (see below and the 'Alphabetic Listing' section of the 'Glossary.pdf' document). gCOLOR can be used to specify the current pen colour and gSETPENWIDTH to specify the pen width. The default is 4-colour mode, as for the default window.

For the default window, the special command DEFAULTWIN is required to change colour modes because that window is automatically created for you in 4-colour mode; DEFAULTWIN clears the default window and resets colour mode to that specified. All other windows must be **created** in the colour mode in which they are required: it may not be changed once they are created.

Once a window has been created with a certain colour mode, colours specified by gCOLOR work in the exactly the same way as in the default window.

❸ Each window can be created with a grey plane if required, in which case gGREY is used to specify whether the black plane, the grey plane or both should be used for all subsequent graphics commands until the next call to gGREY, exactly as described in the first half of this section.

For the default window, the special command DEFAULTWIN is required to enable grey because that window is automatically created for you with only a black plane; DEFAULTWIN 1 closes the default window and creates a new one which has a grey plane. All other windows must be **created** with a grey plane if grey is required.

Once a window has been created with a grey plane, grey is used in precisely the same way as in the default window with grey enabled: `gGREY 0` directs all drawing to the black plane only, `gGREY 1` to the grey plane only and `gGREY 2` to both planes. `gGREY 1` and `gGREY 2` raise an error if the current window does not have a grey plane.

gGREY, gGMODE, gTMODE, gFONT and gSTYLE can all be used with created windows in exactly the same way as with the default window, as described earlier. **They change the settings for the current window only; all the settings are remembered for each window.**

CREATING NEW WINDOWS

The gCREATE function sets up a new window on the screen. It returns an ID number for the window. Whenever you want to change to this window, use gUSE with this ID.

❺ You can create a window with any of the three colour modes by specifying the last optional parameter to gCREATE.

Here is an example using gCREATE and gUSE, contrasting the point 20,20 in the created window with 20,20 in the default window.

```
PROC windows:
   LOCAL id%
   id%=gCREATE(60,40,320,30,1,2)          REM 16-colour mode
   gBORDER 0 :gAT 20,20 :gLINEBY 0,0
   gPRINT " 20,20 (new)"
   GET
   gUSE 1 :gAT 20,20 :gLINEBY 0,0
   gPRINT " 20,20 (default)"
   GET
   gUSE id%
   gCOLOR $88,$88,$88                     REM mid grey
   gPRINT " Back"
   gCOLOR 0,0,0                           REM black
   gPRINT " (with 16 colours)"
   GET
ENDP
```

The line `id%=gCREATE(60,40,320,30,1,2)` creates a window with its top left corner at 60,40 on the screen. The window is set to be 320 pixels wide and 30 pixels deep. (You can use any integer values for these arguments, even if it creates the window partially or even totally off the screen.) The fifth argument to gCREATE specifies whether the window should immediately be visible or not; 0 means invisible, 1 (as here) means visible. The sixth argument specifies the colour mode; 0 means 2-colour mode, 1 means 4-colour mode and 2 (as here) means 16 colour mode. If the sixth argument is not supplied at all (e.g. `id%=gCREATE(60,40,320,30,1)`) the window will have the default 4-colour mode.

Note that 64 drawables (including the default window) may be open at any time, although it is recommended that you use as few windows as possible at any one time. Eight would be a sensible maximum number of windows in practice, although bitmaps may also be used in addition to windows.

❸ You can create a window with only a black plane or with both a black and a grey plane. You cannot create a window with just a grey plane.

Here is an example using gCREATE and gUSE, contrasting the point 20,20 in the created window with 20,20 in the default window.

```
PROC windows:
   LOCAL id%
   id%=gCREATE(60,40,240,30,1,1)
   gBORDER 0 :gAT 20,20 :gLINEBY 0,0
   gPRINT " 20,20 (new)"
   GET
   gUSE 1 :gAT 20,20 :gLINEBY 0,0
   gPRINT " 20,20 (default)"
   GET
   gUSE id%
   gGREY 1                              REM draw grey
   gPRINT " Back"
   gGREY 0
   gPRINT " (with grey)"
   GET
ENDP
```

The line `id%=gCREATE(60,40,240,30,1,1)` creates a window with its top left corner at 60,40 on the screen. The window is set to be 240 pixels wide and 30 pixels deep. (You can use any integer values for these arguments, even if it creates the window partially or even totally off the screen.) The fifth argument to gCREATE specifies whether the window should immediately be visible or not; 0 means invisible, 1 (as here) means visible. The sixth argument specifies whether the window should have a grey plane or not; 0 means black only, 1 (as here) means black and grey. If the sixth argument is not supplied at all (e.g. `id%=gCREATE(60,40,240,30,1)`) the window will not have a grey plane.

gCREATE automatically makes the created window the current window, and sets the current position in it to 0,0. It returns an ID number for this window, which in this example is saved in the variable `id%`.

The `gBORDER 0` command draws a border one pixel wide around the current window. Here this helps show the position and size of the window. (gBORDER can draw a variety of borders. You can even display the 3-D style borders seen in menus and dialogs, with the gXBORDER keyword.)

The program then sets the pixel at 20,20 in this new window, using `gLINEBY 0,0`.

`gUSE 1` goes back to using the default window. The program then shows 20,20 in this window.

Finally, `gUSE id%` goes back to the created window again, and a final message is displayed, in grey and black.

Note that **each window has its own current position**. The current position in the created window is remembered while the program goes back to the default window. **All the other settings, such as the font, style and grey setting are also remembered.**

CLOSING WINDOWS

When you've finished with a particular window, close it with gCLOSE followed by its ID - for example, `gCLOSE 2`. You can create and close as many windows as you like, as long as there are only 64 (8 on the Series 3c) or fewer open at any one time.

If you close the current window, the default window (ID=1) becomes current.

An error is raised if you try to close the default window.

WHEN WINDOWS OVERLAP

Windows can overlap on the screen, or even hide each other entirely. Use the gORDER command to control the foreground/background positions of overlapping windows.

`gORDER 3,1` sets the window whose ID is 3 to be in the foreground. This guarantees that it will be wholly visible. `gORDER 3,2` makes it second in the list; unless the foreground window overlaps it, it too will be visible.

Any position greater than the number of windows you have is interpreted as the end of the list. `gORDER 3,9` will therefore always force the window whose ID is 3 to the background, behind all others.

Note in particular that making a window the current window with gUSE does not bring it to the foreground. You can make a background window current and draw all kinds of things to it, but nothing will happen on the screen until you bring it to the foreground with gORDER.

When a window is first created with gCREATE it always becomes the foreground window as well as the current window.

HIDING WINDOWS

If you are going to use several drawing commands on a particular window, you may like to make it invisible while doing so. When you then make it visible again, having completed the drawing commands, the whole pattern appears on the screen in one go, instead of being built up piece by piece.

Use `gVISIBLE ON` and `gVISIBLE OFF` to perform this function on the current window. You can also make new windows invisible as you create them, by using 0 as the fifth argument to the gCREATE command, and you can hide windows behind other windows.

THE GRAPHICS CURSOR IN WINDOWS

To make the graphics cursor appear in a particular window, use the CURSOR command with the ID of the window. It will appear flashing at the current position in that window, provided it is not obscured by some other window.

The window you specify does not have to be the current window, and does not become current; you can have the cursor in one window while displaying graphical text in another. If you want to move to a different window and put the graphics cursor in it, you must use both gUSE and CURSOR.

Since the default window always has an ID of 1, `CURSOR 1` will, as mentioned earlier, put the graphics cursor in it.

CURSOR OFF turns off the cursor, wherever it is.

INFORMATION ABOUT YOUR WINDOWS

You don't have to keep a complete copy of all the information pertaining to each window you use. These functions return information about the current window:

- gIDENTITY returns its ID number.
- gRANK returns its foreground/background position, from 1 to 8.
- gWIDTH and gHEIGHT return its size.
- gORIGINX and gORIGINY return its screen position.
- ❺ gINFO32 returns information about the font, style, colour setting, overwrite modes and cursor in use.
- ❸ gINFO returns information about the font, style, grey setting, overwrite modes and cursor in use.
- gX and gY return the current position.

OTHER WINDOW KEYWORDS

- gSETWIN changes the position, and optionally the size, of the current window.

You **can** use this command on the default window, if you wish, but you must also use the SCREEN command to ensure that the *text window* (the area for PRINT commands to use) is wholly contained within the default window. See 'The text and graphics windows', later in this section.

- gSCROLL scrolls all or part of both black and grey planes of the current window.
- gPATT fills an area in the current window with repetitions of another window, or with a shaded pattern.
- gCOPY copies an area from another window into the current window, or from one position in the current window to another.

On the Series 5, it is unadvisable to use gCOPY to copy from windows as it is very slow. It should only be used for copying from bitmaps to windows or other bitmaps.

- gSAVEBIT saves part or all of a window as a *bitmap file*.

❸ If a window has a grey plane, the planes are saved as two bitmaps to the same file with the black plane saved first and the grey plane saved next. gLOADBIT, described later, can be used to load bitmap files.

- gPEEKLINE reads back a horizontal line of data in the specified mode (for the Series 5: see the 'Alphabetic Listing'), or from either the black or grey plane (on the Series 3c) of a specified window.

❸ COPYING GREY BETWEEN WINDOWS

The commands gCOPY and gPATT can use two windows and therefore special rules are needed for the cases when one window has a grey plane and the other does not.

With `gGREY 0` in the destination window, only the black plane of the source is copied.

With `gGREY 1` in the destination window, only the grey plane of the source is copied, unless the source has only one plane in which case that plane is used as the source.

With `gGREY 2` in the destination window, if the source has both planes, they are copied to the appropriate planes in the destination window (black to black, grey to grey); if the source has only one plane, it is copied to **both** planes of the destination.

ADVANCED GRAPHICS

This section should provide a taste of some of the more exotic things you can do with OPL graphics.

BITMAPS

A *bitmap* is an area in memory which acts just like an off-screen window. You can create bitmaps with gCREATEBIT.

❺ It is possible to create bitmaps in any of the three colour modes by specifying the optional third argument when using gCREATEBIT. The default is 2-colour mode. Note, however, that black and white bitmaps differ from black and white windows. In particular, if you draw in 16 colours to a black and white bitmap then greys appear as dithered black and white, whereas if you draw exactly the same to a 2-colour graphics window you just get dark greys mapped to black and light greys mapped to white. This enables grey printing on black and white printers.

A further benefit is that the file size will be smaller if the bitmap is saved, with just 1 bpp used for black and white bitmaps.

Note that 64 drawables (including the default window) may be open at any time. Although, as mentioned above, using lots of windows should be avoided in practice, you can sensibly use as many bitmaps as you need up to the maximum.

❸ Note that a bitmap does not have two planes so that gGREY cannot be used.

Bitmaps have the following uses:

- You can manipulate an image in a bitmap before copying it with gPATT or gCOPY to a window on the screen. This is generally faster than manipulating an image in a hidden window.
- You can load *bitmap files* into bitmaps in memory using gLOADBIT, then copy them to on-screen windows using gCOPY or gPATT.

❸ If a black and grey window was saved to file as two bitmaps using gSAVEBIT, you must load them separately into two bitmaps in memory, and copy them one at a time to the respective planes of a window.

OPL treats a bitmap as the equivalent of a window in most cases:

- Both are identified by ID numbers. Only one window or bitmap is current at any one time, set by gUSE.
- If you use bitmaps as well as windows, the **total** number must be 64 (8 on the Series 3c) or fewer.
- The top left corner of the current bitmap is still referred to as 0,0, even though it is not on the screen at all.

Together, windows and bitmaps are known as *drawables* - places you can draw to.

Most graphics keywords can be used with bitmaps in the same way as with windows, but remember that a bitmap corresponds to only one plane in a window. Once you have drawn to it, you might copy it to the appropriate plane of a window.

The keywords that can be used with bitmaps include: gLINEBY, gLINETO, gBOX, gFILL, gCIRCLE (Series 5 only), gELLIPSE (Series 5 only), gCOLOR (Series 5 only), gSETPENWIDTH (Series 5 only), gUSE, gBORDER, gCLOSE, gCLS, gCOPY, gGMODE, gFONT, gIDENTITY, gPATT, gPEEKLINE, gSAVEBIT, gSCROLL, gTMODE, gWIDTH, gHEIGHT, gINFO32 (Series 5) and gINFO (Series 3). These keywords are described earlier in this section.

❺ MASKS

There are several keywords that require an understanding of *masks*. In some cases the mask is a bitmap file, e.g. gBUTTON (see earlier in this section and also the 'Alphabetic Listing' section of the 'Glossary.pdf' file) and for ICON (see the 'Advanced.pdf' document), and in some cases it is an integer containing a *bit-mask*, e.g. for POINTERFILTER (see again 'Advanced.pdf' document).

In all these cases, however, the principle is the same. The pixels or bits which are set in the mask specify pixels or bits in some other argument which are to be used. Pixels and bits which are clear in the mask specify pixels and bits that are not to be used from the other argument.

For example, when using gBUTTON with identical bitmap and mask, cleared pixels on the bitmap are drawn in the background colour of the button (i.e. they are clear) while set pixels are drawn on the button as they appear on the bitmap. This is generally how buttons on Series 5 toolbars appear.

SPEED IMPROVEMENTS

The Psion's screen is usually updated whenever you display anything on it. gUPDATE OFF switches off this feature. The screen will be updated as few times as possible, although you can force an update by using the gUPDATE command on its own. (An update is also forced by GET, KEY and by all graphics keywords which return a value, other than gX, gY, gWIDTH and gHEIGHT).

This can result in a considerable speed improvement in some cases. You might, for example, use `gUPDATE OFF`, then a sequence of graphics commands, followed by `gUPDATE`. You should certainly use `gUPDATE OFF` if you are about to write exclusively to bitmaps.

`gUPDATE ON` returns to normal screen updating.

❸ As mentioned previously, a window with both black and grey planes takes longer to move or scroll than a window with only a black plane. So avoid creating windows with unnecessary grey planes.

Also, remember that scrolling and moving windows require every pixel in a window to be redrawn.

The gPOLY command draws a sequence of lines, as if by gLINEBY and gMOVE commands. If you have to draw a lot of lines (or dots, with `gLINEBY 0,0`), gPOLY can greatly reduce the time taken to do so.

DISPLAYING A RUNNING CLOCK

gCLOCK displays or removes a running clock showing the system time. The clock can be digital or conventional, and can use many different formats. See the 'Alphabetic Listing' section of the 'Glossary.pdf' document for full details.

USER-DEFINED FONTS AND CURSORS

If you have a user-defined font you can load it into memory with gLOADFONT.

❺ gLOADFONT returns a file ID, which can be used only with gUNLOADFONT. The maximum number of font files which may be loaded at any one time is 16. To use the fonts in a loaded font you need to use the UIDs specified in the font file itself.

❸ This returns an ID for the font; use this with gFONT to make the font current. The gUNLOADFONT command removes a user-defined font from memory when you have finished using it.

You can use four extra arguments with the CURSOR command. Three of these specify the ascent, width and height of the cursor. The ascent is the number of pixels (-128 to 127) by which the top of the cursor should be above the baseline of the current font. The height and width arguments should both be between 0 and 255. For example, `CURSOR 1,12,4,14` sets a cursor 4 pixels wide by 14 high in the default window (ID=1), with the cursor top at 12 pixels above the font baseline.

If you do not use these arguments, the cursor is 2 pixels wide, and has the same height and ascent as the current font.

By default the cursor has square corners, is black and is flashing. Supply the fifth argument as 2 for non-flashing or 4 for grey (1 for a rounded cursor is also available on the Series 3c). You can add these together - e.g. use 6 for a grey, non-flashing cursor.

Note that the gINFO32 and gINFO (Series 5 and Series 3c respectively) command returns information about the cursor and font.

THE TEXT AND GRAPHICS WINDOWS

PRINT displays mono-spaced text in the *text window*. You can change the text window font (i.e. that used by PRINT) using the FONT keyword.

❺ You can use any of those fonts listed in Const.oph; see the ‘Calling Procedures’ for an explanation of how to use this file and Appendix E in the ‘Appends.pdf’ document for a listing of it. Initially Courier 11 is used on the Series 5.

It should be noted that when using the console keywords PRINT, AT, SCREEN, etc. the use of Series 5 proportional fonts such as Arial and Times may produce some unexpected behaviour because it is assumed in all cases that mono-spaced font is being used. The reason for this is so that use of keywords such as AT, SCREEN in the console is independent of whether the font is proportional or monospaced. An example of this behaviour may be seen when using inverted text: the rectangle to invert is calculated assuming that the font is mono-spaced, and hence the area inverted is larger than the text printed when using a proportional font. Another example is that a new line will be used before it appears necessary when using a proportional font, since the number of characters which will fit on a line is also calculated assuming the font is mono-spaced.

❸ You can use any of those fonts listed earlier in this section in the description of gFONT; initially font 4 is used on the Series 3c.

The text window is in fact part of the default graphics window. If you have other graphics windows in front of the default window, they may therefore hide any text you display with PRINT.

Initially the text window is very slightly smaller than the default graphics window which is full-screen size. They are not the same because **the text window height and width always fits a whole number of characters of the current text window font**. If you use the FONT command to change the font of the text window, this first sets the default graphics window to the maximum size that will fit in the screen (excluding any status window on the Series 3c) and then resizes the text window to be as large as possible inside it.

You can also use the STYLE keyword to set the style for all characters subsequently written to the text window. This allows the mixing of different styles in the text window.

❸ You can only use those styles which do not change the size of the characters - i.e. inverse video and underline. (Any other styles will be ignored.)

Use the same values as listed for gSTYLE, earlier in this section.

To find out exactly where the text window is positioned, use `SCREENINFO info%()`. This sets `info%(1)/info%(2)` to the number of pixels from the left/top of the default window to the left/top of the text window. (These are called the margins.) `info%(7)` and `info%(8)` are the text window's character width and height respectively.

The margins are fully determined by the font being used and therefore change from their initial value only when FONT is used. You cannot choose your own margins. gSETWIN and SCREEN do not change the margins, so you can use FONT to select a font (also clearing the screen), followed by SCREENINFO to find out the size of the margins with that font, and finally gSETWIN and SCREEN to change the sizes and positions of the default window and text window taking the margins into account (see example below). The margins will be the same after calling gSETWIN and SCREEN as they were after FONT.

It is not generally recommended to use both the text and graphics windows. Graphics commands provide much finer control over the screen display than is possible in the text window, so it is not easy to mix the two.

If you do need to use the text window, for example to use keywords like EDIT, it's easy to use SCREEN to place it out of the way of your graphics windows. You can, however, use it on top of a graphics window - for example, you might want to use EDIT to simulate an edit box in the graphics window. Use gSETWIN to change the **default window** to about the size and position of the desired edit box. The text window moves with it - you must then make it the same size, or slightly smaller, with the SCREEN command. Use 1,1 as the last two arguments to SCREEN, to keep its top left corner fixed. `gORDER 1,1` will then bring the default window to the front, and with it the text window. EDIT can then be used.

Here is an example program which uses this technique - moving an 'edit box', hiding it while you edit, then finally letting you move it around.

```
PROC gsetw1:
    LOCAL a$(100),w%,h%,g$(1),factor%,info%(10)
    LOCAL margx%,margy%,chrw%,chrh%,defw%,defh%
    SCREENINFO info%()                REM get text window info
    margx%=info%(1) :margy%=info%(2)
    chrw%=info%(7) :chrh%=info%(8)
    defw%=23*chrw%+2*margx%           REM new default window width
    defh%=chrh%+2*margy%              REM ... and height
    w%=gWIDTH :h%=gHEIGHT
    gSETWIN w%/4+margx%,h%/4+margy%,defw%,defh%
    SCREEN 23,1,1,1                   REM text window
    PRINT "Text win:"; :GET
    gCREATE(w%*0.1,h%*0.1,w%*0.8,h%*0.8,1)          REM new window
    gPATT -1,gWIDTH,gHEIGHT,0         REM shade it
    gAT 2,h%*0.7 :gTMODE 3
```

```
    gPRINT “Graphics window 2”
    gORDER 1,0                REM back to default+text window
    EDIT a$                   REM you can see this edit
    gORDER 1,65               REM to background - could use 9 for Series 3c
    CLS
    a$=“”
    PRINT “Hidden:”;
    GIPRINT “Edit in hidden edit box”
    EDIT a$                   REM YOU CAN’T SEE THIS EDIT
    GIPRINT “”
    gORDER 1,0 :GET           REM now here it is
    gUSE 1                    REM graphics go to default window
    DO                        REM move default/text window around
       CLS
       PRINT “U,D,L,R,Quit”;
       g$=UPPER$(GET$)
       IF KMOD=2              REM Shift key moves quickly
         factor%=10
       ELSE
         factor%=1
       ENDIF
       IF g$=“U”
         gSETWIN gORIGINX,gORIGINY-factor%
       ELSEIF g$=“D”
         gSETWIN gORIGINX,gORIGINY+factor%
       ELSEIF g$=“L”
         gSETWIN gORIGINX-factor%,gORIGINY
       ELSEIF g$=“R”
         gSETWIN gORIGINX+factor%,gORIGINY
       ENDIF
    UNTIL g$=“Q” OR g$=CHR$(27)
ENDP
```

FRIENDLIER INTERACTION

Everyday OPL programs can use the same graphical interface seen throughout the Psion:

- **Menus offer lists of options for you to choose from. You can also select these options with shortcut keys like Ctrl+A, Ctrl+B (Psion-A, Psion-B on the Series 3c) etc.**
- **Dialogs let a program ask for all kinds of information numbers, filenames, dates and times etc. in one go.**
- **Screen messages such as 'Busy' are available.**
- **On the Series 3c, the status window is also available.**

Menu keywords begin with an `M`, and dialog keywords with a `D`. In this manual a lower case is used for these letters for example, `mINIT` and `dEDIT` but you can type them using upper or lower case letters.

MENUS

Menus provide a simple way for any reasonably complex OPL program to let you choose from its various options.

To display menus in OPL generally takes three basic steps:

- Use the mINIT command. This prepares OPL for new menus.
- Use the mCARD command (and the mCASC command on the Series 5) to define each menu.
- Use the MENU function to display the menus.

You use the displayed menus like any others on the Psion. Use the arrow keys to move around the menus. Press Enter or an option's shortcut key (or on the Series 5, tap with the pen) to select an option, or press Esc to cancel the menus without making a choice. In either case, the menus are removed, the screen redrawn as it was, and MENU returns a value to indicate the selection made.

DEFINING THE MENUS

The first argument to mCARD is the name of the menu. This will appear at the top of the menu; the names of all of the menus form a bar across the top of the screen.

From one to eight options on the menu may be defined, each specified by two arguments. The first is the option name, and the second the keycode for a shortcut key. This specifies a key which, when pressed together with the Ctrl key (Psion key on the Series 3c), should select the option. (Your program must still handle shortcut keys which are pressed without using the menu.) It is easiest to specify the shortcut key with `%` e.g. `%a` gives the value for `a`.

If an upper case character is used for the shortcut key keycode, the Shift key must be pressed as well to select the option (on the Series 5, the shortcut key will appear as 'Shift+Ctrl+A' for example). If you supply a keycode for a lower case character, the option is selected only **without** the Shift key pressed. Both upper and lower case keycodes for the same character can be used in the same menu (or set of menus). This feature may be used to increase the total number of shortcut keys available, and is also commonly used for related menu options e.g. `%m` (`%z` on the Series 3c) might be used for zooming to a larger font and `%M` (`%Z`) for zooming to a smaller font (as in the built-in applications).

For example,

```
mCARD "Comms","Setup",%s,"Transfer",%t
```

defines a menu with the title `Comms`. When you move to this menu using ◂ or ▸, you'll see it has the two options `Setup` and `Transfer`, with shortcut keys Ctrl+S and Ctrl+T (Psion-S and Psion-T on the Series 3c) respectively (and no Shift key required). On the other hand,

```
mCARD "Comms","Setup",%S,"Transfer",%T
```

would give these options the shortcut keys Shift+Ctrl+S and Shift+Ctrl+T (Shift-Psion-S and Shift-Psion-T on the Series 3c).

The options on a large menu may be divided into logical groups (as seen in many of the menus for the built-in applications) by displaying a line under the final option in a group. To do this, you must pass the negative value corresponding to the shortcut key keycode for the final option in the group. For example, `-%A` specifies shortcut key Shift+Ctrl+A (Shift-Psion-A on the Series 3c) and displays a grey line under the associated option in the menu.

Each subsequent mCARD defines the next menu to the right. A large OPL application might use mCARD like this:

```
mCARD “File”,“New”,%n,“Open”,%o,“Save”,%s
mCARD “Edit”,“Cut”,%x,“Copy”,%c,“Paste”,-%v,“Eval”,%e
mCARD “Search”,“First”,%f,“Next”,%g,“Previous”,%p
```

5 ADDITIONAL FEATURES ON THE SERIES 5

On the Series 5, more advanced menu features are available in addition to the more basic features described above. These are as follows:

- menu items without shortcuts
- menu items that are dimmed
- menu items with checkboxes
- menu items with option buttons (sometimes known as radio buttons)
- cascaded menus
- popup menus (see ‘Displaying menus’ below).

It is possible to have menu items without shortcuts, by specifying shortcut values between 1 and 32. The value specified is still returned if the item is selected.

Dimming, checkboxes and option buttons are controlled by adding the following values to the shortcut keycode (the constants are found in Const.oph. See the ‘Calling Procedures’ section of the ‘Basics.pdf’ document for details of how to use this file and Appendix E in the ‘Appends.pdf’ document for a listing of it.):

constant name	*value*	*effect*
KMenuDimmed%	$1000	item dimmed
KMenuCheckBox%	$0800	item has checkbox
KMenuOptionStart%	$0900	item starts option button list
KMenuOptionMiddle%	$0A00	item in middle of option button list
KMenuOptionEnd%	$0B00	item ends option button list
KMenuSymbolOn%	$2000	symbol on (checkbox/option button)
KMenuSymbolIndeterminate%	$4000	symbol indeterminate

Dimming a menu item makes it unavailable and on trying to select it, the info print ‘This item is not available’ is automatically displayed. Items with checkboxes have a tick symbol on or off on their left hand side to show whether or not they have been selected. The start, middle and end option buttons are for specifying a group of related items that can be selected exclusively (i.e. if one item is selected then the others are deselected). The number of middle option buttons is variable.

Adding in the `KMenuSymbolOn%` flag sets the tick on a checkbox or the button on an option button item on. The display of ticks and option buttons is automatically changed appropriately when you select one of these items, but your program needs to maintain the state of any checkbox or option button between displays of the menu.

A single menu card can have more than one set of option buttons and checkboxes, but option buttons in a set should be kept together. For speed, OPL does not check the consistency of these items’ specification.

Cascaded items on which less important menu items can be displayed may also be created. The cascade must be defined before use in a menu card. The following is an example of a 'Bitmap' cascade under the File menu of a possible OPL drawing application.

```
mCASC "Bitmap","Load",%L,"Merge",%M
mCARD "File","New",%n,"Open",%o,"Save",%s,"Bitmap>",16,"Exit",%e
```

The trailing > character specifies that a previously defined cascade item is to be used in the menu at this point: it is not displayed in the menu item. A cascade has a filled arrow head displayed along side it in the menu. The cascade title in mCASC is also used only for identification purposes and is not displayed in the cascade itself. This title needs to be identical to the menu item text apart from the >. For efficiency, OPL doesn't check that a defined cascade has been used in a menu and an unused cascade will simply be ignored.

Shortcut keys used in cascades may be added with the appropriate constant values to enable checkboxes, option buttons and dimming of cascade items.

As is typical for cascade titles, a shortcut value of 16 is used in the example above. This prevents the display or specification of any shortcut key. However, it is possible to define a shortcut key for a cascade title if required, for example to cycle through the options available in a cascade.

DISPLAYING THE MENUS

The MENU function displays the menus defined by mINIT and mCARD, and waits for you to select an option. It returns the shortcut key keycode of the option selected, in the case supplied by you, whether you used Enter or the shortcut key itself (or the pen on the Series 5) to select it. If you supplied a negative shortcut key keycode for an underlined option, it is converted to its positive equivalent.

If you cancel the menus by pressing Esc, MENU returns 0.

❸ When a set of menus is displayed, the highlight is positioned to the menu and option that the user selected previously (or, if no menus have previously been displayed, to the first option in the first menu).

This works only if your program has only one set of menus. If you have another set of menus, the cursor is **still** set to the position of the menu and option selected in the first set of menus (if that position exists in the new menus).

To avoid this confusion on the Series 3c or to maintain the position of the highlight across menu calls on the Series 5, use `m%=menu(init%)` and set `init%` to zero the first time a set of menus is displayed. The cursor will in this case be positioned to the first option in the first menu. `init%` is set to a value which specifies the menu and option selected, and should be passed to MENU the next time that same set of menus is called If your program has more than one set of menus, you should have a different `init%` variable for each set of menus.

❺ DISPLAYING A POPUP MENU

A popup menu which appears at a specified point on the screen, may also be defined and drawn using mPOPUP. **Note that popup menus have only one pane and need not and should not be within the mINIT...MENU structure.** mPOPUP returns the value of the shortcut code in the same way as MENU. For example,

```
mPOPUP(0,0,0,"Continue",%c,"Exit",%e)
```

The first two arguments specify the position of one corner and the third argument specifies which corner this is. This third argument takes values as follows,

	corner
0	top left
1	top right
2	bottom left
3	bottom right

Thus, the example above specifies a popup menu with `0,0` as its top left-hand corner and with the items ‘Continue’ and ‘Exit’, with the shortcuts Ctrl+C and Ctrl+E respectively.

You can add the same values to the shortcut key keycode as those used with mCARD and mCASC to display dimmed items, checkboxes and option buttons. Note, however, that cascades in popup menus are not supported. For example,

```
mPOPUP(0,0,0,“View1”,%v OR $2900,“View2”,%b OR $A00,“View3”,%n OR $B00)
```

would display a popup menu with option buttons, with the symbol initially set on on the `View1` item (`$2000` is ORed into it as well as `$900`).

PROBLEMS WITH MENUS

You must ensure that you do not use the same shortcut key twice when defining the menus, as OPL does not check for this.

Each menu definition uses some memory, so ‘No system memory’ errors are possible.

Don’t forget to use mINIT before you begin defining the menus.

If the menu titles defined by mCARD are too wide in total to fit on the screen (wider than 40 characters on the Series 5), MENU will raise a ‘Too wide’ error.

❺ Shortcut values **must** be alphabetic character codes or numbers between the values of 1 and 32. Any other values will raise an ‘Invalid arguments’ error.

Note also that on the Series 5, a menu is discarded when an item fails to be added successfully. In effect the previous mINIT statement is discarded together with any previous mCARD statements. This avoids the problem of trying to use a badly constructed menu item.

It is therefore incorrect to ignore mCARD errors by having an ONERR label around an mCARD call (see the ‘Errors.pdf’ document for more details). If you do, the menu is discarded and a ‘Structure fault’ will then be raised on using mCARD or MENU without first using mINIT again.

❸ When choosing shortcut keys, do not use those such as the number keys which produce different characters when used with the Psion key. Unless you have a good reason not to, stick with `a` to `z` and `A` to `Z`.

A MENU EXAMPLE

This procedure allows you to press the Menu key and see a menu. You might instead be typing a number or some text into the program, or moving around in some way with the arrow keys, and this procedure returns any such keypresses. You could use this procedure instead of a simple GET whenever you want to allow a menu to be shown, and its shortcut keys to work.

Each of the options in the menus have a corresponding procedure named `proc` plus the shortcut key letter so for example, the option with shortcut key Ctrl+N (Psion-N) is handled by the procedure `procn`.

This procedure uses the technique of calling procedures by strings, as described in the 'Advanced.pdf' document.

❺
```
PROC kget%:
  LOCAL k%,h$(9),a$(5)
  h$="nosciefgd"                          REM our shortcut keys
  WHILE 1
     k%=GET
     IF k%=$122                           REM Menu key
          mINIT
          mCARD "File","New",%n,"Open",%o,"Save",%s
          mCARD "Edit","Copy",%c,"Insert",-%i,"Eval",%e
          mCARD "Search","First",%f,"Next",%g,"Previous",%d
          k%=MENU
          IF k% AND (LOC(h$,CHR$(k%))<>0)  REM MENU CHECK
               a$="proc"+CHR$(k%)
               @(a$):                     REM procn:, proco:, ...
          ENDIF                        REM END OF MENU CHECK
     ELSEIF KMOD AND $4              REM shortcut key pressed directly?
          k%=k%+$40                       REM remove Ctrl modification
          IF LOC(h$,CHR$(k%))             REM DIRECT SHORTCUT KEY CHECK
               a$="proc"+CHR$(k%)
               @(a$):                     REM procn:, proco:, ...
          ENDIF                       REM END OF DIRECT SHORTCUT KEY CHECK
     ELSE                             REM some other key
          RETURN k%
     ENDIF
  ENDWH
ENDP
```

❸
```
PROC kget%:
  LOCAL k%,h$(9),a$(5)
  h$="nosciefgd"                          REM our shortcut keys
  WHILE 1
     k%=GET
     IF k%=$122                           REM Menu key
          mINIT
          mCARD "File","New",%n,"Open",%o,"Save",%s
          mCARD "Edit","Copy",%c,"Insert",-%i,"Eval",%e
          mCARD "Search","First",%f,"Next",%g,"Previous",%d
          k%=MENU
          IF k% AND (LOC(h$,CHR$(k%))<>0)       REM MENU CHECK
```

```
                a$="proc"+CHR$(k%)
                @(a$):                    REM procn:, proco:, ...
            ENDIF                       REM END OF MENU CHECK
        ELSEIF k% AND $200              REM shortcut key pressed directly?
            k%=k%-$200                    REM remove Psion key code
            IF LOC(h$,CHR$(k%))           REM DIRECT SHORTCUT KEY CHECK
                a$="proc"+CHR$(k%)
                @(a$):                    REM procn:, proco:, ...
            ENDIF                       REM END OF DIRECT SHORTCUT KEY CHECK
        ELSE                            REM some other key
            RETURN k%
        ENDIF
    ENDWH
ENDP
```

`procn:`, `proco:`, etc would need to be specified to use this example in practice on any of the machines:

```
PROC procn:
...
ENDP

PROC proco:
...
ENDP
...
```

Note that this procedure allows you to press a shortcut key with or without the Shift key. So Ctrl+Shift+N would be treated the same as Ctrl+N, similarly on the Series 3c, Shift-Psion-N would be treated the same as Psion-N.

Neither LOC nor the @ operator (for calling procedures by strings) differentiate between upper and lower case. If you have Shifted shortcut keys you will need to compare against two sets of shortcut key lists. For example, with shortcut keys `%A`, `%C`, `%a` and `%d`, you would have upper/lowercase shortcut key lists like `hu$="AC"` and `hl$="ad"`, and the "MENU CHECK" section becomes:

```
IF k%<=%Z                           REM if upper case shortcut key
    IF LOC(hu$,CHR$(k%))
      a$="procu"+CHR$(k%)
      @(a$):                        REM procua:, procuc:, ...
    ENDIF
ELSE                                REM else lower case shortcut key
    IF LOC(hl$,CHR$(k%))
      a$="procl"+CHR$(k%)
      @(a$):                        REM procla:, procld:, ...
    ENDIF
ENDIF
```

(This calls procedures `procua:`, `procuc:`, `procla:` and `procld:`). If a shortcut key was pressed directly you cannot tell from `k%` whether Shift was used; so make the same change to the "DIRECT SHORTCUT KEY CHECK" section, but use `IF KMOD AND 2` instead of `IF k%<=%Z`.

DIALOGS

In OPL, dialogs are constructed in a similar way to menus:

- Use the dINIT command to prepare OPL for a new dialog. If you give a string argument to dINIT it will be displayed as a title for the dialog. On the Series 5, the title will be in a grey box at the top of the dialog and on the Series 3c it will be separated from the rest of the dialog by a horizontal line.
- Define each line of the dialog, from top to bottom. There are separate commands for each type of item you can use in a dialog for example, dEDIT for editing a string, dDATE for typing in a date, and so on.
- Use the DIALOG function to display the dialog. In general it returns a number indicating the line you were on when you pressed Enter (counting any title line as line 1), or 0 if you pressed Esc.

Use the up and down arrow keys to move from line to line, and enter the relevant information, as in any other Psion dialog. You can even press Tab to produce vertical lists of options when appropriate.

Each of the commands like dEDIT and dDATE specifies a variable to take the information you type in. If you press Enter to complete the dialog, the information is saved in those variables. The dialog is then removed, and the screen redrawn as it was.

You can press Esc to abandon the dialog without making any changes to the variables.

If you enter information which is not valid for the particular line of the dialog, you will be asked to enter different information.

Here is a simple example. It assumes a global variable `name$` exists:

```
PROC getname:
    dINIT "Who are you?"
    dEDIT name$,"Name:"
    DIALOG
ENDP
```

This procedure displays a dialog with `Who are you?` as its top-line title, and an edit box for typing in your name. If you end by pressing Enter, the name you have typed will be saved in `name$`; if you press Esc, `name$` is not changed.

When the dialog is first displayed, the existing contents of `name$` are used as the string to edit.

Note that the dialog is automatically created with a width suitable for the item(s) you defined, and is centred in the screen.

LINES YOU CAN USE IN DIALOGS

This section describes the various commands that can define a line of a dialog. In all cases:

- `prompt$` is the string which will appear on the left side of the line.
- `var` denotes an argument which **must** be a LOCAL or GLOBAL variable, **because it takes the value you enter**. Single elements of arrays may also be used, but not field variables or procedure parameters. (var is just to show you where you must use a suitable variable you don't actually type var.)

  Where appropriate, this variable provides the initial value shown in the dialog.

Although examples are given using each group of commands, you can mix commands of any type to make up your dialog.

More details of the commands may be found in the 'Alphabetic Listing' section of the 'Glossary.pdf' document.

STRINGS, SECRET STRINGS AND FILENAMES

`dEDIT var str$,prompt$,len%`

defines a string edit box.

`len%` is an optional argument. If supplied, it gives the width of the edit box (allowing for the widest possible character in the font). The string will scroll inside the edit box, if necessary. If `len%` is not supplied, the edit box is made wide enough for the maximum width `str$` could be. (You may wish to set a suitably small `len%` to stop some dialogs being drawn all the way across the screen)

`dXINPUT var str$,prompt$`

defines a secret string edit box, such as for a password. A special symbol will be displayed for each character you type, to preserve the secrecy of the string.

`dFILE var str$,prompt$,f%`

❺ `dFILE var str$,prompt$,f%,Uid1&,Uid2&,Uid3&`

defines a filename editor or selector box. dFILE automatically has ‘Folder’ and ‘Disk’ selectors (only a ‘Disk’ selector on the Series 3c) on the lines below it. `f%` controls whether you have a file editor or selector in your dialog, and the kind of input allowed. On the Series 5, files selected may also be restricted by UID. See dFILE in the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document for full details of how use dFILE.

Here is an example dialog using these commands:

```
PROC info:
    LOCAL n$(30),pw$(16),f$(255)
    dINIT “Your personal info”
    dEDIT n$,“Name:”,15
    dXINPUT pw$,“Password:”
    dFILE f$,“Log file:”,0
    RETURN DIALOG
ENDP
```

On the Series 5, you may want to replace the dFILE line with the following:

`dFILE f$,“Log file,Folder,Disk”,0`

as default prompts for the folder and disk selector boxes are not provided as on the Series 3c.

This procedure returns ‘True’ if Enter was used, indicating that the GLOBAL variables `n$`, `pw$` and `f$` have been updated.

❺ `dEDITMULTI var ptrData&,prompt$,widthInChars%,noOfLines%,maxLen%`

defines a multi-line edit box to go into a dialog. Normally the resulting text would be used in a subsequent dialog, saved to file or printed using the Printer OPX (see the ‘OPX.pdf’ document). The use of this dialog command is more complicated than the others (see the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document for full details).

CHOOSING ONE OF A LIST

```
dCHOICE var choice%,prompt$,list$
```

defines a choice list. `list$` should contain the possible choices, separated by commas for example, `"Yes,No"`. The `choice%` variable specifies which choice should initially be shown 1 for the first choice, 2 for the second, and so on.

For example, here is a simple choice dialog:

```
PROC dcheck:
    LOCAL c%
    c%=2                    REM default to "View2"
    dINIT "Change View"
    dCHOICE c%,"View:","View1,View2,View3"
    IF DIALOG               REM returns 0 if cancelled
       ...                  REM change view
    ENDIF
ENDP
```

❺ On the Series 5, extended choice lists may also be defined by using more than one dCHOICE statement (see the 'Alphabetic Listing' section of the 'Glossary.pdf' document for full details of how to do this).

❺ `dCHECKBOX chk%,prompt$`

creates a checkbox entry. This is similar to a choice list with two items, except that the list is replaced by a checkbox with the tick either on or off. The state of the checkbox is maintained across calls to the dialog. Initially you should set the live variable `chk%` to 0 to set the tick symbol off and to any other value to set it on. `chk%` is then automatically set to 0 if the box is unchecked or -1 if it is checked when the dialog is closed.

NUMBERS, DATES AND TIMES

```
dLONG var long&,prompt$,min&,max&
```

and

```
dFLOAT var fp,prompt$,min,max
```

define edit boxes for long integers and floating-point numbers respectively. Use dFLOAT to allow fractions, and dLONG to disallow them. `min(&)` and `max(&)` give the minimum and maximum values which are to be allowed. There is no separate command for ordinary integers use dLONG with suitable `min&` and `max&` values.

```
dDATE var long&,prompt$,min&,max&
```

and

```
dTIME var long&,prompt$,type%,min&,max&
```

define edit boxes for dates and times. `min&` and `max&` give the minimum and maximum values which are to be allowed.

For dDATE, `long&`, `min&` and `max&` are specified in "days since 1/1/1900". The DAYS function is useful for converting to "days since 1/1/1900".

For dTIME, `long&`, `min&` and `max&` are in “seconds since 00:00”. The DATETOSECS and SECSTODATE functions are useful for converting to and from “seconds since midnight” (they actually use “seconds since 00:00 on 1/1/1970”).

dTIME also has a `type%` argument. This specifies the type of display required:

`type%`	*time display*
0	absolute time without seconds
1	absolute time with seconds
2	duration without seconds
3	duration with seconds

❺ Two additional types are also available on the Series 5.

4	absolute times in 24 hour clock
8	time not displaying hours

For example, `03:45` is an *absolute time*, while 3 hours 45 minutes is a *duration.*

This procedure creates a dialog, using these commands:

```
PROC delivery:
    LOCAL d&,t&,num&,wt
    d&=DAYS(DAY,MONTH,YEAR)
    DO
      t&=secs&:
    UNTIL t&=secs&:
    num&=1 :wt=10
    dINIT “Delivery”
    dLONG num&,“Boxes”,1,1000
    dFLOAT wt,“Weight (kg)”,0,10000
    dDATE d&,“Date”,d&,DAYS(31,12,1999)
    dTIME t&,“Time”,0,0,DATETOSECS(1970,1,1,23,59,59)
    IF DIALOG                          REM returns 0 if cancelled
      ...                              REM rest of code
    ENDIF
ENDP

PROC secs&:
    RETURN HOUR*INT(3600)+MINUTE*60
ENDP
```

The `secs&:` procedure uses the HOUR and MINUTE functions, which return the time as kept by the Psion. It is called twice to guard against an incorrect result, in the (albeit rare) case where the time ticks past the hour between calling HOUR and calling MINUTE.

The INT function is used in `secs&:` to force OPL to use long integer arithmetic, avoiding the danger of an ‘Overflow’ error.

`d&` and `t&` are set up to give the current date and time when the dialog is first displayed. The value in `d&` is also used as the minimum value for dDATE, so that in this example you cannot set a date before the current date.

DATETOSECS is used to give the number of seconds representing the time 23:59. The first three arguments, `1970`, `1` and `1`, represent the first day from which DATETOSECS begins calculating.

RESULTS FROM DDATE

dDATE returns a value as a number of days. To convert this to a date:

❺ use DAYSTODATE which converts a number of days since 1/1/1990, to the corresponding date. See the 'Alphabetic Listing' for full details.

❸

- If you are dealing only with days on or after 1/1/1970, you can subtract 25567 (DAYS(1,1,1970)), multiply by 86400 (the number of seconds in a day), and use SECSTODATE.
- To handle days before 1/1/1970 as well, you can call the Operating System to perform the conversion. This procedure is passed one parameter, the number of days, and from it sets four global variables day%, month%, year% and yrdy%. It calls the Operating System with the OS function:

```
PROC daytodat:(days&)
   LOCAL dyscent&(2),dateent%(4)
   LOCAL flags%,ax%,bx%,cx%,dx%,si%,di%
   dyscent&(1)=days&
   si%=ADDR(dyscent&()) :di%=ADDR(dateent%())
   ax%=$0600                    REM TimDaySecondsToDate fn.
   flags%=OS($89,ADDR(ax%))          REM TimManager int.
   IF flags% AND 1
      RAISE (ax% OR $ff00)
   ELSE
      year%=PEEKB(di%)+1900
      month%=PEEKB(UADD(di%,1))+1
      day%=PEEKB(UADD(di%,2))+1
      yrdy%=PEEKW(UADD(di%,6))+1
   ENDIF
ENDP
```

If you do use this procedure, be careful to type it exactly as shown here.

Displaying text

`dTEXT prompt$,body$,type%`

defines `prompt$` to be displayed on the left side of the line, and `body$` on the right. **There is no variable associated with dTEXT.** If you use a null string (`""`) for `prompt$`, `body$` is displayed across the whole width of the dialog.

`type%` is an optional argument. If specified, it controls the alignment of `body$`:

`type%`	*effect*
0	left align `body$`
1	right align `body$`
2	centre `body$`

❺ Note that alignment of `body$` is only supported when `prompt$` is null, with the body being left aligned otherwise.

In addition, you can add any or all of the following three values to `type%`, for these effects:

$200	draw a line below this item.
$400	make the prompt (not the body) selectable.
$800	make this item a text separator

dTEXT is not just for displaying information. Since DIALOG returns a number indicating the line you were on when you pressed Enter (or 0 if you pressed Esc), you can use dTEXT to offer a choice of options, rather like a menu:

```
PROC select:
  dINIT “Select action”
  dTEXT “Add”,“”,$402
  dTEXT “Copy”,“”,$402
  dTEXT “Review”,“”,$402
  dTEXT “Delete”,“”,$402
  RETURN DIALOG
ENDP
```

In each case `type%` is `$402` ($400+2). The `$400` makes each prompt selectable, allowing you to move the cursor on to it. Note that **only** the prompts are selectable: if you try the example given for the Series 3c below on the Series 5, you will see that the items are **not** selectable because the prompt is null. However, the items will be centre aligned in the dialog.

❸ In addition, you can add any or all of the following three values to `type%`, for these effects:

$100	use bold text for body$.
$200	draw a line below this item.
$400	make this line selectable. It will also be bulleted if prompt$ is not “”.

dTEXT is not just for displaying information. Since DIALOG returns a number indicating the line you were on when you pressed Enter (or 0 if you pressed Esc), you can use dTEXT to offer a choice of options, rather like a menu:

```
PROC select:
  dINIT “Select action”
  dTEXT “”,”Add”,$402
  dTEXT “”,”Copy”,$402
  dTEXT “”,”Review”,$402
  dTEXT “”,”Delete”,$402
  RETURN DIALOG
ENDP
```

In each case `type%` is `$402` ($400+2). The `$400` makes each text string selectable, allowing you to move the cursor on to it, while `2` makes each string centred.

See the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document for full details of dTEXT.

DISPLAYING EXIT KEYS

Most dialogs are completed by pressing Enter to confirm the information typed, or Esc to cancel the dialog. These keys are not usually displayed as part of the dialog.

However, some Psion dialogs offer you a simple choice, by showing pictures of the keys you can press. A simple “Are you sure?” dialog might, for example, show the two keys ‘Y’ and ‘N’, and indicate the one you press.

If you want to display a message and offer Enter, Esc and/or Space as the exit keys, you can display the entire dialog with the ALERT function.

If you want to use other keys, such as `Y` and `N`, or display the keys below other dialog items such as dEDIT, create the dialog as normal and use the dBUTTONS command to define the keys.

ALERT and dBUTTONS are explained in detail in the ‘Alphabetic listing’ section of the ‘Glossary.pdf’ document.

OTHER DIALOG INFORMATION

POSITIONING DIALOGS

If a dialog overwrites important information on the screen, you can position it with the dPOSITION command. Use dPOSITION at any time between dINIT and DIALOG.

dPOSITION uses two integer values. The first specifies the horizontal position, and the second, the vertical. `dPOSITION -1,-1` positions to the top left of the screen; `dPOSITION 1,1` to the bottom right; `dPOSITION 0,0` to the centre, the usual position for dialogs.

`dPOSITION 1,0`, for example, positions to the right-hand edge of the screen, and centres the dialog half way up the screen.

5 OTHER DIALOG FEATURES

On the Series 5, dINIT can take a second optional parameter to specify additional dialog features. This may be any ORed combination of the following constants:

value	*effect*
1	buttons positioned on the right rather than at the bottom
2	no title bar (any title in dINIT is ignored)
4	use the full screen
8	don’t allow the dialog box to be dragged
16	pack the dialog densely (not buttons though)

Constants for these flags are supplied in Const.oph. See the ‘Calling Procedures’ section of the ‘Basics.pdf’ document for details of how to use this file and Appendix E in the ‘Appends.pdf’ document for a listing of it.

It should be noted that dialogs without titles cannot be dragged regardless of the “No drag” setting. Dense packing enables more lines to fit on the screen for larger dialogs.

For example, the following could be used for a large dialog:

```
dINIT “Series 5 Dialog”,$15
```

so that the dialog covers the full screen, has buttons (as defined by dBUTTONS) on the right and has items densely packed.

RESTRICTIONS ON DIALOGS

The following general restrictions apply to all dialogs:

- Only one dialog may be in use at a time.
- A dialog must be initialised (dINIT), defined (dEDIT etc.) and displayed (DIALOG) in the same procedure.

❺

- No error is raised if the dialog is too wide or too long to fit on the screen: it is up to the programmer to ensure the dialog is displayed in a suitable way.

❸

- A dialog may consist of up to nine lines, including any title. Filename editors count as two lines, and exit keys count as three. A ‘Too many items’ error is raised if this limit is exceeded.
- If the width of any line would make the dialog too wide, a ‘Too wide’ error is raised when DIALOG is called.

SERIES 5 TOOLBAR USAGE

The toolbar on the Series 5 replaces the Series 3a, 3c and Siena status window. All Series 5 OPL programs should use the ROM module `Z:\System\Opl\Toolbar.opo` to create a toolbar window with a title, four buttons and a clock.

The public interface to `Toolbar.opo` is supplied in `Z:\System\Opl\Toolbar.oph` which is reproduced below. The procedures and their usage are then discussed in detail.

TOOLBAR.OPH

```
REM Toolbar.oph version $100
REM Header file for OPL's toolbar
REM (c)Copyright Psion PLC 1997

REM Public procedures
EXTERNAL TBarLink:(appLink$)
EXTERNAL TBarInit:(title$,scrW%,scrH%)
EXTERNAL TBarSetTitle:(name$)
EXTERNAL TBarButt:(shortcut$,pos%,text$,state%,bit&,mask&,flags%)
EXTERNAL TBarOffer%:(winId&,ptrType&,ptrX&,ptrY&)
EXTERNAL TBarLatch:(comp%)
EXTERNAL TBarShow:
EXTERNAL TBarHide:

REM The following are global toolbar variables usable by OPL programs
REM or libraries. Usable after toolbar initialisation:
REM TbWidth%  the pixel width of the toolbar
REM TbVis%    -1 if visible and otherwise 0
REM TbMenuSym%current Show toolbar menu symbol (ORed with shortcut)

REM Flags for toolbar buttons
CONST KTbFlgCmdOnPtrDown%=$01
```

```
CONST KTbFlgLatchStart%=$12      REM start of latchable set
CONST KTbFlgLatchMiddle%=$22     REM middle of latchable set
CONST KTbFlgLatchEnd%=$32        REM end of latchable set
CONST KTbFlgLatched%=$04         REM set for current latched item in set

REM End of Toolbar.oph
```

TYPICAL TOOLBAR.OPO USAGE

Typically a program would use Toolbar.opo in the following way:

- LOADM "Z:\System\Opl\Toolbar.opo" to load the toolbar module. This module must remain loaded as long as you need to use the toolbar.
- 'Link' the toolbar-specific globals into the top-level of your program, using TBarLink:
- Initialise the toolbar, creating an invisible toolbar window with title and clock, using TBarInit:
- Add normal or latchable toolbar buttons to the toolbar, using TBarButt:
- Make the toolbar visible, using TBarShow:
- Offer all pointer events to the toolbar so that it can call your commands, change the system clock type or display the task list, using TBarOffer%:
- Latch a button down to represent the current view when the view changes, using TBarLatch:
- Change the toolbar title to the current document name, using TBarSetTitle:
- Show and hide the toolbar as appropriate, using TBarShow: and TBarHide:
- Provide *command-handling* procedures to be called by `Toolbar.opo` when a toolbar button is pressed.

TBARLINK:

Usage: `TBarLink:(appLink$)`

TBarLink: provides all toolbar globals required in your application. It has to be called before TBarInit: and from a higher level procedure in your application than the one in which the globals are used.

`appLink$` is the name of the so-called *continuation procedure* in your main application. TBarLink: calls this procedure, which should then go on to run the rest of your program. This allows the globals declared in TBarLink: to exist until the application exits. `appLink$` must represent a procedure with name and parameters like:

```
PROC appContTBarLink:
    REM Continue after 'linking' toolbar globals
    myAppInit:     REM run rest of program
    ...
ENDP
```

i.e. taking no parameters and with no return-type specification character, so that it can be called using `@(appLink$):.`

TBARINIT:

Usage: `TBarInit:(title$,scrW%,scrH%)`

Called at start of application only, this procedure creates the toolbar window, which guarantees that there will be sufficient memory available to display the toolbar at any subsequent time. The toolbar is made invisible when not shown. This procedure also draws all toolbar components except the buttons.

Note that, for speed, TBarInit: turns graphics auto-updating off (using gUPDATE OFF). If automatic updating is required, use gUPDATE ON after TBarInit: returns.

Note also that TBarInit: sets compute mode off (see SETCOMPUTEMODE: in the 'System OPX' section of the 'OPX.pdf' document) allowing the program to run at foreground priority when in foreground. By default OPL programs have compute mode on (i.e. they run at background priority even when in foreground).

`title$` is the title shown in the toolbar. You should change this to the name of your current file, using TBarSetTitle:.

`scrW%` is the full-screen width (gWIDTH at startup).

`scrH%` is the full-screen height (gHEIGHT at startup).

TBARSETTITLE:

Usage: `TBarSetTitle:(name$)`

Sets the title in the toolbar.

`name$` is the name of your current file for file-based applications (i.e. applications with the `APP...ENDA` construct containing `FLAGS 1`), or the name of your application for non-file applications.

TBARBUTT:

Usage: `TBarButt:(shortcut$,pos%,text$,state%,bit&,mask&,flags%)`

Adds a button to the previously initialised toolbar.

`shortcut$` is the command shortcut for your application, which is used by `Toolbar.opo` to perform the command when a toolbar button is selected. On selecting the toolbar button, `Toolbar.opo` calls your procedure to perform the required command or action. `shortcut$` is case sensitive in the sense that `Toolbar.opo` calls your procedure named:

- `"cmd"+shortcut$+%:` for unshifted, lower-case shortcuts,
- `"cmdS"+shortcut$+%:` for shifted, upper-case shortcuts.

For example, if you have the following two commands that also have associated toolbar buttons:

```
mCARD "View","DoXXX",%x,"DoYYY",%Y    REM shortcuts Ctrl+X,Shift+Ctrl+Y
```

you would need to provide command-handling procedures:

- ```
  PROC cmdX%:
  REM handle shortcut Ctrl+X (lower case shortcut$="x")
  ```
- ```
  PROC cmdSY%:
  REM handle shortcut Shift+Ctrl+Y (upper case shortcut$="Y")
  ```

You would create buttons using, e.g.

- ```
  TBarButt:("x",1,"DoXXX",0,XIcon&,XMask&,0)
  REM Button for calling cmdX:
  ```

- ```
  TBarButt:("Y",1,"DoYYY",0,YIcon&,YMask&,0)
  REM Button for calling cmdSY:
  ```

`pos%` is the button position, with `pos%=1` for the top button.

`text$` and `state%` take values as required for gBUTTON.

`bit&` and `mask&` are the button's icon bitmap and mask, used in in the same way as for gBUTTON. Note that a button is a purely graphical entity and so doesn't own the bitmaps. Therefore the bitmaps may not be unloaded while the button is still in use.

`flags%` lets you control how the button is used in two distinct and mutually exclusive ways, as follows:

1. Two *latchable* buttons are often used by the built-in applications to indicate the current view. For an example, see the latchable 'Desk' and 'Sci' buttons in the built-in Calc application.
   A set of latchable toolbar buttons can be specified in TBarButt: by setting flags% to one of:
   KTbFlgLatchStart% for the first button in the latchable set.
   KTbFlgLatchMiddle% for any middle buttons (these are optional and not generally used).
   KTbFlgLatchEnd% for the last button in the latchable set.
   To latch a button down initially to represent the initial setting, OR KTbFlgLatched% with one of the above settings. E.g.
   TBarButt:(sh1$,pos%,txt1$,st%,bit1&,msk1&,KTbFlgLatchStart%)
   TBarButt:(sh2$,pos%,txt2$,st%,bit2&,msk2&,KTbFlgLatchEnd% OR TbFlgLatched%)
   will latch down the second button in the set initially.
   In the toolbar window, the button with KTbFlgLatchStart% set must be above the buttons (if any) with KTbFlgLatchMiddle% set, and these in turn must be above the button with KTbFlgLatchEnd%.
   Only one button in a set is ever latched and pressing another button unlatches the one that was previously set. After pressing and releasing a previously unlatched button in a latchable set, Toolbar.opo will, as usual, call your command-handling procedure. When the command has succeeded in changing view, this procedure should set the new state of the button by calling TBarLatch:(comp%) where comp% is the button number to be latched. This will also unlatch any button that was previously latched. The example below shows how a 'View1' button press, with 'v' as shortcut, should be handled. The other latchable button in this set might be 'View2' with shortcut 'w':

   ```
   PROC cmdV%:
      IF SetView1%:=0              REM if no error
           TBarLatch:(KView1TbarButton%) REM your CONST KView1TBarButton%
           CurrentView%=1
      ENDIF
   ENDP

   PROC cmdW%:
      IF SetView2%:=0              REM if no error
           TBarLatch:(KView2TbarButton%) REM your CONST KView2TBarButton%
           CurrentView%=2
      ENDIF
   ENDP
   ```

   You should call the same command-procedures when the command is performed via a menu or via a keyboard shortcut. This will ensure that the button is latched as required.

2. A setting of flags% can also be used to specify that your procedure should be called when the toolbar button is tapped (rather than when the button is released, which is the default). The ‘Go to’ button in the Program editor works in this way, displaying the popup list of procedures when the button is touched. To implement this using TBarButt:, pass flags%=KTbFlgCmdOnPtrDown% and provide a procedure named: “cmdTbDown”+shortcut$+%:
which could provide a popup menu, as follows:

```
PROC cmdTbDownC%:
    REM popup next to button, with point specifying
    REM top right corner of popup
    IF mPOPUP(ScrWid%-TbWidth%, 97, KMPopupPosTopRight%, “Cancel”,
                  0,”Clear”,%c)
        cmdC%:          REM Do the command itself
     ENDIF
ENDP
```

In this case the shortcut is not case-sensitive. Note that when this flag is used, the menu command-procedure is not used directly because a popup is not required when the command is invoked via the menu or via a keyboard shortcut.

TBAROFFER%:

Usage: `TBarOffer%:(winId&,ptrType&,ptrX&,ptrY&)`

Offers a pointer event to the toolbar, returning -1 if used and 0 if not used. If not used, the event is available for use by your application.

It is important to call this procedure whenever you receive a pointer event, even when the event is not in the toolbar window, thus enabling `Toolbar.opo` to release the current button, both visually and otherwise.

TBarOffer%: handles:

- the tapping of the clock to change the type between analog and digital system-wide.
- the tapping of the toolbar title to display the list of open files.
- the selection of toolbar buttons, calling your command procedures.
- drawing of the button in the appropriate state.

As usual, the word ‘pointer’ indicates a pen on the Series 5.

`winId&` is the ID of the window that received the pointer event.

`ptrType&` is pointer event type as returned by GETEVENT32 (pen up, pen down or drag).

`ptrX&,ptrY&` is co-ordinate of pointer event.

TBARLATCH:

Usage: `TBarLatch:(button%)`

Latches down a toolbar button, where `button%=1` for the top button in the toolbar. TBarLatch: also unlatches any button in the latchable set that was previously latched. See TBarButt: for further details on latching buttons.

TBARSHOW:

Usage: `TBarShow:`

Makes the toolbar visible. The toolbar must exist before calling this procedure. Use TBarInit: to create an invisible toolbar with no buttons. Use TBarButt: to add buttons.

TBARHIDE:

Usage: `TBarHide:`

Makes the toolbar invisible.

PUBLIC TOOLBAR GLOBALS

The following toolbar globals, provided by TBarLink:, may be used in Series 5 OPL applications:

- TbWidth% is the pixel width of the toolbar. The rest of the screen width is available for the application.
- TbVis% is -1 if visible and otherwise 0
- TbMenuSym% is the current 'Show toolbar' menu symbol to be added to menu shortcut for View|Show toolbar, e.g.
  `mCard "View","Show toolbar",%t or TbMenuSym%`
  `TbMenuSym%=(KMenuCheckBox% OR KMenuSymbolOn%)` if the Toolbar is visible and `TbMenuSym%=KMenuCheckBox%` if invisible. The menu item will therefore be a checkbox item, with the check present or not as appropriate.

GIVING INFORMATION

❸ STATUS WINDOW TEMPORARY AND PERMANENT

Pressing Psion-Menu when an OPL program is running will always display a temporary status window. This status window is in front of all the OPL windows, so your program can't write over it.

Use `STATUSWIN ON` or `STATUSWIN ON,type%` to display a permanent status window. It will be displayed until you use `STATUSWIN OFF`. `type%` specifies the status window type.

❸ The small status window is displayed for `type%=1` and the large status window either when `type%` is not supplied or when `type%=2`.

Siena There is only one type of status window which will be displayed whatever `type%` you use.

You might use `STATUSWIN ON` when Control-Menu is pressed, for consistency with the rest of the Series 3c.

The status window is displayed on the right-hand side of the screen.

❸ THE RANK OF THE STATUS WINDOW

Important: The permanent status window is **behind** all other OPL windows. In order to see it, you **must** use either FONT or both SCREEN and gSETWIN, to reduce the size of the text window and the default graphics window. You should ensure that your program does not create windows over the top of it.

FONT automatically resizes these windows to the maximum size excluding any status window. It should be called after creating the status window because the new size of the text and graphics windows depends on the

size of the status window. Note that `FONT -$3fff,0` leaves the current font and style - it just changes the window sizes and clears them.

If you use SCREEN and gSETWIN instead of FONT, you should use the STATWININFO keyword (described next) to find out the size of the status window.

FINDING THE POSITION AND SIZE OF A STATUS WINDOW

`curtype%=STATWININFO(type%,extent%())` sets the four element array `extent%()` as follows:

`extent%(1)` = pixels from left of screen to status window

`extent%(2)` = pixels from top of screen to status window

`extent%(3)` = status window width in pixels

`extent%(4)` = status window height in pixels for status window `type%`.

`type%=3` specifies the compatibility mode status window and `type%=-1` specifies whichever type of status window is currently shown. Otherwise, use the same values of `type%` as for STATUSWIN.

STATWININFO returns the type of the **current** status window. The values are as for `type%`, or zero if there is no current status window.

If type%=-1 for the current status window and there is none, STATWININFO returns consistent information in extent%() corresponding to a status window of width zero and full screen height positioned one pixel to the right of the physical screen.

So to set a graphics window to have height `h%` and to use the full screen width up to the current status window (if any), but leaving a one pixel gap between the graphics window and the status window, you could use:

```
STATWININFO(-1,extent%()) :gSETWIN 0,0,extent%(1),h%
```

Alternatively you could simply use `FONT -$3fff,0` as described under STATUSWIN above, which also sets the height to full screen height and sets the text window size to fit inside it.

WHAT THE STATUS WINDOW DOES

The status window always displays the OPL program name, a clock and, by default, an icon. This will be the default OPL icon, unless your program is an *OPA* with its own icon. (*OPA*s are described in the 'Advanced.pdf' document.) In addition, the settings selected in the 'Status window' menu option of the System screen are automatically used in OPL status windows. The status window will, therefore, also display all the indicators required, and a digital or analog clock as selected there.

The status window is inaccessible to, and does not affect, the OPL keywords gORDER and gRANK.

You can set or change the name displayed in the status window with SETNAME for example, `SETNAME "ABCD"` or `SETNAME a$`.

USING A DIAMOND LIST IN THE STATUS WINDOW

Your program may have several distinct modes/views/screens between which you would like the diamond key to switch. The built-in applications use the diamond key extensively Agenda uses it to switch to the different views, while Word switches between 'Normal' and 'Outline' view.

The diamond list is displayed in the status window. It is a list of modes, views or screens which are stepped through as the diamond key is pressed.

OPL programs can set up a diamond list. Use

```
DIAMINIT pos%,str1$,str2$,...
```

to initialise the list (this discards any existing list). The list can be initialised before or after a status window is displayed.

`str1$`, `str2$` etc. contain the text to be displayed in the status window for each item in the list.

`pos%` is the initial item on to which the diamond indicator should be positioned, with `pos%=1` specifying the first item. (Any value greater than the number of strings specifies the final item.)

If `pos%=0`, or if DIAMINIT is used on its own with no arguments, no bar is defined.

If `pos%=-1` the list is replaced by the icon instead in the large status window.

If `pos%>=1` **you must supply at least this many strings**.

Defining a list uses some memory, so 'No system memory' errors are possible.

`DIAMPOS pos%` positions the diamond indicator in a list. You might move the indicator to the next item when the diamond key is pressed and to the previous item when Shift+the diamond key is pressed. The diamond key has keycode value 292 and KMOD returns 2 when the Shift key is pressed.

Positioning outside the range of the items wraps around in the appropriate way if there are three items in the list, `DIAMPOS 4` positions to the first.

`DIAMPOS 0` causes the diamond symbol to disappear.

Use chr$(4) to display a diamond key in a menu. If you use it as a shortcut key, a Shift will be added automatically.

INFORMATION MESSAGES

GIPRINT displays an information message for 2 seconds, in the bottom right corner of the screen. For example, `GIPRINT "Not Found"` displays `Not Found`. If a string is too long for the screen, it will be clipped.

You can add an integer argument to control the corner in which the message appears:

value	*corner*
0	top left
1	bottom left
2	top right
3	bottom right

❺ Constants for these corner values are supplied in Const.oph. See the 'Calling Procedures' section of the 'Basics.pdf' document for details of how to use this file and Appendix E in the 'Appends.pdf' document for a listing of it.

For example, `GIPRINT "Who?",0` prints `Who?` in the top left corner.

Only one message can be shown at a time. You can make the message go away for example, if a key has been pressed with `GIPRINT ""`.

'BUSY' MESSAGES

Messages which say a program is temporarily busy, or cannot respond for some reason, are by convention shown in the bottom left corner. The BUSY command lets you display your own messages of this sort. Use `BUSY OFF` to remove it.

`BUSY "Paused..."`, for example, displays `Paused...` in the bottom left corner. This remains shown until `BUSY OFF` is used.

You can control the corner used in the same way as for GIPRINT.

You can also add a third argument, to specify a delay time (in half seconds) before the message should be shown. Use this to prevent BUSY messages from continually appearing very briefly on the screen.

For example, `BUSY "Wait:",1,4` will display `Wait:` in the bottom left corner after a delay of 2 seconds. As soon as your program becomes responsive to the keyboard, it should use `BUSY OFF`. If this occurs within two seconds of the original BUSY, no message is seen.

The maximum string length of a BUSY message is 80 characters (on the Series 3c, 19 characters) and an 'Invalid argument' error is returned for any value in excess of this.

Only one message can be shown at a time.

INDEX

SYMBOLS

A

B

C

D

F

G

S

T

U

W