OPL

User Guide

OPL

INTRODUCTION

Welcome to the OPL User Guide for Psion Series 5, Series 3c and Siena machines. This User Guide describes how you can create and run your own programs using your Psion’s built-in programming language.

KEY

The OPL User Guide uses the following symbols to indicate points of special interest:

✍ A **note**, for any additional information or details.

⚠ A **warning**, for advice on avoiding problems.

DIFFERENCES BETWEEN MACHINES

It is important to note that the versions of OPL on the various machines covered by this User Guide are not exactly the same. When information is given which is specific to one of the machines you will see one of the following icons:

❺	for the Series 5
❸	for the Series 3c
Siena	for the Siena

These icons most frequently appear next to paragraphs which describe machine-dependent features. However, in some places they appear along side section headings, in which case that section and all its subsections are specific to the machine indicated. Where no indication is given that information is machine-dependent, you should assume it applies to all three machines.

In general, the Series 3c and Siena versions of OPL are the same, except in a few small points, and hence where you see the Series 3c icon and where information given is indicated as being specific to the Series 3c, it generally refers to **both the Series 3c and the Siena**, unless specific reference is made to the contrary.

CONTENTS

This User Guide is divided into a number of sections according to subject matter and the level of programming expertise expected of the user. Each of the sections is a self-contained PDF file with its own examples and index. Below is a list of the User Guide sections together with their filenames in brackets and an outline of their contents.

If you haven't programmed in OPL before, it is recommended that you work through the Basics section first. This will help you to get to grips with the basic concepts involved.

Use the Overview and Alphabetical Listing section at any time to find either the appropriate command for the task you want to carry out, or the precise usage of a given command. The information given on commands in this section will also tell you where you can read more about them.

- **You can jump straight to any of the documents listed below by clicking on its title.**

BASICS (BASICS.PDF)

This serves as an introduction to programming in OPL and explains its basic concepts. It is divided into 4 sections:

- Creating & Running Programs.
- Variables & Constants.
- Loops & Branches.
- Calling Procedures.

DATA FILE & DATABASE HANDLING (DATABASE.PDF)

This covers the database capabilites of OPL. Its two sections cover:

- Creating, saving and performing operations on data files

and

- Using databases on the Series 5.

GRAPHICS & FRIENDLIER INTERACTION (GUI.PDF)

This document is divided into 2 sections, dealing with:

- Using the powerful graphics capabilities of OPL

and

- How to make your programs friendlier by using features such as menus and dialogs.

OPL

OPL & DISKS (DISKS.PDF)

This document provides information on OPL’s handling on disks used in Psion machines.

EXAMPLE PROGRAMS (PROGRAMS.PDF)

This section provides supplementary example programs for all the machines covered by the User Guide. These programs aim to illustrate a wide range of OPL’s capabilities.

ERROR HANDLING (ERRORS.PDF)

This document contains information on dealing with errors in your OPL programs, including:

- Syntax errors.
- Errors that occur when running programs.

This document also contains a list of errors and their likely causes.

ADVANCED TOPICS (ADVANCED.PDF)

This document covers a range of topics suitable for more advanced programmers. These include:

- Where files are stored.
- Safe pointer arithmetic.
- Creating OPL applications.
- Cacheing procedures on the Series 3c and Siena.
- Sprite handling on the Series 3c and Siena.
- I/O functions and commands.
- Recording and playing sounds.
- DYL handling.
- Memory Allocation.

USING OPXS ON THE SERIES 5 (OPX.PDF)

This section provides information on OPX procedures provided in the Series 5 ROM for handling the following:

- Date / time extras.
- System controls.
- Bitmaps.
- Sprites.
- Database extras.
- Printing.

OVERVIEW & ALPHABETICAL LISTING (GLOSSARY.PDF)

- The Overview section of this document lists all the OPL keywords grouped according to purpose. Use this section to find the appropriate command for what you want to do.
- The Lisitng section lists the OPL keywords alphabetically, providing a reference guide to the usage of each.

APPENDICES A-G (APPENDS.PDF)

The seven appendices in this section cover the following information:

- **Appendix A:** Summary for experienced users.
- **Appendix B:** Operators and Logical expressions.
- **Appendix C:** Serial/Parallel ports and printing.
- **Appendix D:** Character codes.
- **Appendix E:** Listing of CONST.OPH for the Series 5.
- **Appendix F:** SQL specification for the Series 5.
- **Appendix G:** EPOC32 error values (Series 5 only).

OPL

BASICS

This part of the OPL User Guide introduces the basic concepts of programming in OPL. It is divided into 4 sections:

- **Creating & Running Programs: this covers the stages of entering, translating and running a program in OPL.**
- **Variables & Constants: this section explains the way values are stored and handled in OPL.**
- **Loops & Branches: this section covers how to repeat commands, wait for given conditions and so on.**
- **Calling Procedures: this explains how to link several parts of a program together.**

CONTENTS

CREATING & RUNNING PROGRAMS

There are 3 stages to producing a program using OPL, the Psion programming language:

- **Type in the program, using the Program editor.**
- **Translate the program. This makes a new version of your program in a format which can "run".**
- **Run the program. If it does not work as you had intended, re-edit it, then translate and run it again.**

This section guides you through these stages with a simple example. If you wish to follow the example, note that each instruction for you to do something is numbered.

CREATING A NEW MODULE

As well as the word *program*, you'll often see the word *module* used. The terms *program* and *module* are used almost interchangeably to describe each OPL file - you say "OPL **module**" like you might say "Word Processor **document**".

Create a new module and give it a name:

❺

1. Click the 'New File' button (or select 'Create New File' from the 'File' menu).
2. Select 'Program' from the 'Program' selector.
3. Type `test` as the 'Name' to use for this OPL module and press Enter. You will move into the Program editor.

Module names can be up to 256 characters long (including their folder names), like other file names on the Series 5. The names may include any characters **except** `\`, `/` and `:`, and any trailing spaces or dots (.) will be stripped automatically.

❸

1. Move to the Program icon on the System screen, and select 'New file' from the 'File' menu.
2. Type `test` as the name to use for this OPL module and press Enter. You will move into the Program editor.

Module names can be up to 8 characters long, like other filenames on the Series 3c. The names can include numbers, but must start with a letter.

It's always best to choose a name that describes what the module does. Then, when you've written several modules, you can still recognise which is which.

INSIDE THE PROGRAM EDITOR

When you first move into the Program editor you will see that `PROC :` has already been entered on the first line, and `ENDP` on the third.

PROC and ENDP are the *keywords* that are used to mark the start and end of a *procedure*. Larger modules are broken up into procedures, each of which has one specific function to perform. A simple OPL module, like the one you are going to create, consists of only one procedure.

A procedure consists of a number of *statements* — instructions upon which the Psion acts. You type these statements, in order, between `PROC :` and `ENDP`. When you come to *run* the program, the Psion goes through the statements one by one. When the last statement in the procedure has been completed and ENDP is reached, the procedure ends.

You can type and edit in the Program editor in much the same way as in the Word application, except that text you type does not word-wrap; you should press Enter at the end of each statement. Note also that on the Series 3c, the Program editor does not offer text layout features such as styles and emphases.

✍ You can use upper or lower case letters when entering OPL keywords.

AN EXAMPLE PROCEDURE

The next few pages work with this example procedure:

```
PROC test:
  PRINT “This is my OPL program”
  PAUSE 80
  CLS
  PRINT “Press a key to finish”
  GET
ENDP
```

This procedure does nothing of any real use it is just an example of how some common OPL keywords (PRINT, PAUSE, CLS and GET) are used. The procedure first displays This is my OPL program on the screen. After a few seconds the screen is cleared and then Press a key to finish is displayed. Then, when you press a key, the program finishes.

TYPE IN AND EDIT THE PROCEDURE

Before you type the statements that constitute the procedure, you must type a name for it, after the word PROC. The flashing cursor is automatically in the correct place for you to do this (before the colon). You can choose any name you like within the following restrictions:

❺ Procedure names may have up to 32 characters. The alphabetic and numeric characters are allowed and also the underscore character, _. The first character of any procedure name must be either an underscore or an alphabetic character.

❸ Procedure names may have up to 8 characters. The alphabetic and numeric characters are allowed only. The first character of any procedure name must be an alphabetic character.

For simple procedures which are the only procedure in a module, you might use the same filename you gave the module.

Type test . The top line should now read PROC test: .

Press the down arrow key. The cursor is already indented, as if the Tab key had been pressed.

You can now type the statements in this procedure:

Type PRINT “This is my OPL program”. (Note the space after PRINT.) Press Enter at the end of the line.

Each new line is automatically indented, so you don’t need to press the Tab key each time. These indents are not obligatory, though as you’ll see, they can make a procedure easier to read. **However, other spacing, such as the space between PAUSE and 80, is essential for the procedure to work properly.**

Type the other statements in the procedure. Press Enter at the end of each line. You are now ready to translate the module and then run it.

When you are entering the statements in a procedure you can, if you want, combine adjacent lines by separating them with a space and colon. For example, the two lines:

```
PAUSE 80
CLS
```

could be combined as this one line:

```
PAUSE 80 :CLS
```

You can, of course, use the other Psion applications at any time while you are editing an OPL module.

❺ To return to editing your program, e*ither*

- tap on the Program icon on the Extras bar, *or*
- select the module’s name on the System screen, *or*
- use the Task List to return to the Program editor.

❸ Use Control-Word (hold down the Control key and press the Word button) to return to the Program editor to continue editing your program.

WHAT THE KEYWORDS DO WHEN THE PROGRAM RUNS

`PRINT` - takes text you enter between quote marks, and displays it on the screen. The text to be displayed, in the first statement, is `This is my OPL program`.

`PAUSE` - pauses the program, for a specified number of twentieths of a second. `PAUSE 80` waits for 4 seconds. (`PAUSE 20` would wait for 1 second, and so on.)

`CLS` - clears the screen.

`GET` - waits for you to press a key on the keyboard.

TRANSLATING A MODULE

The translation process makes a separate version of your program in a format which the Psion can run.

You’d usually try to translate a module as soon as you finish typing it in, to check for any typing mistakes you’ve made, and then to see if the program runs as you intended.

❺ • Select the ‘Translate’ option from the ‘Tools’ menu *or* tap the ‘Tran’ button on the toolbar menu.

❸ • Select the ‘Translate’ option from the ‘Prog’ menu.

The Series 3c ‘Prog’ menu also has a ‘S3 translate’ option, for translating the current program in a form which can run on a Series 3 (as opposed to a Series 3c).

WHAT HAPPENS WHEN YOU TRANSLATE A MODULE?

First: the procedures in the module are checked for errors

If the Psion cannot understand a procedure, because of a typing error, a message is shown, such as ‘Syntax error’. The cursor is positioned at the point where the error was detected, so that you can correct it. For example, you might have typed `PRONT “This is...”`, or `PAUSE80` without the space.

When you think you’ve corrected the mistake, select ‘Translate’ again. If there is still a mistake, you are again taken back to where it was detected.

⚠ If you’ve already used up almost all of the memory, the Psion may be unable to translate the program, and will report a ‘No system memory’ message. You’ll need to free some memory before trying again.

When ‘Translate’ can find no more errors, the translation will succeed, producing a separate version of your module in a format which the Psion can run.

There may still be errors in your program at this point because there are some errors which cannot be detected until you try to run the program.

RUNNING AFTER TRANSLATING

When your module translates successfully, the ‘Run program’ dialog is displayed, asking whether to run the translated module. You’d usually run it straight away in order to test it.

⚠ Running a module does require some free memory, so again a ‘No system memory’ message is possible.

Press ‘Y’ to run the module; the screen is cleared, and the module runs.

When the module has finished running, you return to the Program editor, with the cursor where it was before.

If an error occurs while the module is running, you will return to editing the module, and the cursor will be positioned at the point where the error occurred.

FILE MANAGEMENT

NEW OPL MODULES

You can create new OPL modules in the same way as new Word documents.

❺ Either create it from the Program editor using the ‘Create New file’ option in ‘File’ menu, or from the System screen by clicking on the ‘New File’ button (or select ‘Create New File’ from the ‘File’ menu).

The module names are listed on the System screen with a Program icon next to them. The Program icon looks like a sheet of paper with “OPL” on it. Successfully translated modules will also be listed in the same folder as their corresponding Program file with the OPL icon next to them. The OPL icon is just the letters “OPL” with a shadow.

To re-edit an existing OPL program, you can open the Program application and use the ‘Open file’ option from the ‘File’ menu. You could also select the file directly from the System screen. This will automatically open the file **and** launch the Program application. Files which launch their associated applications when selected are known as *documents*. The application *UID* (unique identifier) is stored in the document header which is read by the system. As far as the user is concerned, the UID specifies a document’s *type*. A *non-document file* does **not** have an application UID and is displayed on the system screen with a special icon (a question mark) showing that it is unrecognised. Non-document files are known as *external files*.

Opening Program from its icon in the Extras bar will re-open the Program file last in use.

❸ Either create it from the Program editor using the ‘New file’ option in ‘File’ menu, or from the System screen by moving to the Program icon and using its ‘New File’ option.

Your module names are listed below the Program icon. The Program icon is a speech bubble containing “OPL” on a grey background. The word ‘Program’ is shown below the icon if there are no modules at all.

The names under the RunOpl icon are those modules which have been translated successfully. The RunOpl icon is just “OPL” in a speech bubble.

To re-edit an existing OPL program, use the ‘Open file’ option in the Program editor, or move to the Program icon in the System screen and select the filename from the list.

COPYING MODULES

Use the ‘Copy file’ option in the System screen to copy modules (or translated modules). See the User Guide for full details. You can also use the ‘Save as’ option in the Program editor itself, to make new copies of an OPL module.

DELETING MODULES

You can delete an OPL module (or a translated version) as you would any other file. Go to the System screen, move the highlight on to the file and use the ‘Delete file’ option.

❸ If you delete all of your translated modules, the RunOpl icon will remain on the System screen, with the word RunOpl beneath it.

‘FILE IS IN USE’

If you see a “*File*’ is in use’ (‘File or device in use’ on the Series 3c) error message when deleting or copying an OPL module, the file is open — it is currently being edited in the Program editor. Exit the file and then try again.

If it’s the translated file you’re trying to delete or copy, “*File*’ is in use’ (‘File or device in use’ on the Series 3c) means that the translated file is currently running. Stop the running program by going to the running program, then either wait for the program to complete or press Ctrl+Esc (on the Series 5; Psion+Esc on the Series 3c) to stop it, and then you can try again.

MORE ABOUT RUNNING MODULES

RUNNING FROM THE PROGRAM EDITOR

You can run a module at any time from within the Program editor, by selecting ‘Run program’ (‘Run’ on the Series 3c) from the ‘Tools’ menu (‘Prog’ menu on the Series 3c). This runs the **translated** version of your program; if you’ve made changes to the module and haven’t translated it again, you must translate the module again, or the changes have no effect.

‘Run program’ (‘Run’ on the Series 3c) displays a dialog, letting you select the name of **any** translated module which you want to run.

RUNNING MODULES FROM THE SYSTEM SCREEN

The names of any successfully translated programs automatically appear in the System screen.

❺ Translated modules appear in the System screen with the OPL icon to the left of them. They have the same name as the Program file from which they were translated with the extension `.OPO` added to their name, and appear in the same folder as their corresponding Program file. Just move the highlight on to the name of the translated program you want to run, and select it.

❸ Translated modules appear underneath the RunOpl icon. This appears at the right-hand end of the list of icons (past the Program icon), and is usually off the right-hand edge of the screen. Just move the highlight on to the name of the translated program you want to run, and press Enter.

Like the Program editor, RunOpl is assigned a keypress - you can press Control-Calc (hold down Control and press the Calc button) as the short-cut to move to the RunOpl icon, whatever you’re doing. (If there is a running program, this instead moves **directly** to it.)

When an OPL module has been successfully translated and run, you will usually run it from the System screen. While you’re still editing and testing, however, it’s quicker to run it from inside the Program editor. This also positions the cursor for you, if errors occur.

STOPPING A PROGRAM WHILE IT'S RUNNING

❺ **To stop a running program, press Ctrl+Esc.** (If you've gone away from the running program it will still be running, and you must first return to it. This is done by either selecting it from the System screen or by using the list of open files to switch to it. Then Ctrl+Esc will stop it.)

To pause a running program, press Ctrl+Fn+S. It will be paused as soon as it next tries to display something on the screen. **Press Ctrl+Fn+Q to let the program resume running.**

❸ **To stop a running program, press Psion+Esc.** (If you've gone away from the running program it will still be running, and you must first return to it. This is done by pressing Control-Calc and/or selecting it from under the RunOpl icon in the System screen before pressing Psion-Esc.)

To pause a running program, press Control-S. It will be paused as soon as it next tries to display something on the screen. **Press any other key to let the program resume running.**

DISPLAYING A STATUS WINDOW

❺ The Series 5 does not have status windows: it has a toolbar instead. You should see the 'Friendlier Interaction' section of the 'GUI.pdf' document for details of this.

❸ A temporary status window is always available while an OPL program is running. Press Psion-Menu to see it. As you'll see, there are keywords for displaying a status window yourself.

LOOKING AT A RUNNING PROGRAM

❺ If you translate and run a module from the Program editor, the Task list will still allow you to return to the Program editor, even if the translated program has not finished running. A 'Running…' message is shown — you can move the cursor around the program as normal, but you can't edit it.

To return to the running version, either use the Task list or select it from the System screen. It will be in bold, to show that it is currently running.

❸ If you translate and run a module from the Program editor, the Control-Word keypress will still return to the Program editor, even if the translated program has not finished running. A 'Busy' message is shown — you can move the cursor around the program as normal, but you can't edit it.

To return to the running version, select it from beneath the RunOpl icon in the System screen. It will be in bold, at the top of the list, to show that it is currently running. Alternatively, press Control-Calc to return to it.

RUNNING MORE THAN ONE MODULE

If a module is running, and you select a second one from the System screen, the first one is **not** replaced — both modules run together, and will be displayed in bold on the System screen. On the Series 5, you can swap between them using the list of open files, on the Series 3c use Control-Shift-Calc.

MENU OPTIONS WHILE EDITING

While you're typing in the procedure, all the options on the 'Edit' menu such as 'Copy' ('Copy text' on the Series 3c) and 'Paste' ('Insert text' on the Series 3c) - are available and can be used as in Word. Refer to the User Guide for more information.

❺ The menu options available are in general similar to those found in other applications, such as Word. The 'Tools' menu has options for translating and running the current program. It also has a 'Show last error' option, to re-display an error which prevented successful translation, and a 'Preferences' option to determine the fonts available and whether spaces, tabs and paragraph ends are shown in the Program editor. It also provides an 'Infrared' option (see the User Guide for more details of using infrared). The 'Create standard files' option creates files in RAM from ROM files: see the 'Calling Procedures' section of the this document for more details of this.

The 'Format' menu provides an 'Font' dialog for changing fonts and styles in the Program editor. The 'Indentation' option can be used to set the tab width and to turn auto-indentation on and off.

The 'File' menu also include 'Import text' and 'Export as text' options for importing text and exporting as text. These can be used to convert Program files from Series 3a, 3c and Siena to Series 5 and vice versa. To convert from earlier Program files to Series 5 Program files you need to:

1. Create a new Program document.
2. Import the text using the 'Import text' option from the 'More' cascade in the 'File' menu.
3. Translate and run as usual.

⚠ Note that there maybe some incompatibility between Series 5 OPL and earlier versions. See Appendix A in the 'Appends.pdf' document for a summary of these and other documents as appropriate for further details.

The toolbar on the left-hand side of the screen provides easy access via buttons to four options and also a clock. The options are 'Tran' ('Translate'), 'Find', 'Find next' and 'Go to'. These options are all self-explanatory, except perhaps for the last: 'Go to' gives a list (scrolled if necessary) of all the procedure in the module. Selecting one of them jumps to the beginning of the specified procedure.

❸ The menus available are the same as in the Word application, except that the 'Word' menu has been replaced by the 'Prog' menu. The 'Prog' menu has options for translating and running the current program. It also has a 'Show error' option, to re-display an error which prevented successful translation, and an 'Indentation' option, for setting the tab width and to turn auto-indentation on and off in the Program editor.

Unlike Word, the Program editor only ever uses one template for creating new files, called 'default'. When you use the 'New file' option, the 'Use template' line is therefore unavailable; the new file is created using the 'default' template automatically. If you wish to change the 'default' template, you can use the 'Save as template' option to replace it with the current file. **Do not try to swap templates between Word and the Program editor.** 'Set preferences' allows you to choose between bold/normal and mono-spaced/ proportional text. It also has options for showing tabs, spaces, paragraph ends, soft hyphens and forced line breaks.

There is no 'Password' option.

③ THE DIAMOND KEY

The diamond key allows you to switch between a 'Normal' and an 'Outline' view of your OPL module. The 'Outline' view lists only the names of each procedure, for quick navigation around the module.

SUMMARY

❺ Tap the ‘New file’ button on the system screen and select ‘Program’ as the ‘Program’.

Type in your procedure.

Select ‘Translate’ from the ‘Tools’ menu.

When a module translates correctly you are given the option to run it. You can run it again at any time, either with ‘Run program’ in the ‘Tools’ menu, or directly from the System screen.

Use Ctrl+Esc to stop a running program.

Use Ctrl+Fn+S to pause a program and Ctrl+Fn+Q to restart it again.

❸ Move to the Program icon in the System screen and select the ‘New file’ option.

Type in your procedure.

Select ‘Translate’ from the ‘Prog’ menu.

When a module translates correctly you are given the option to run it. You can run it again at any time, either with ‘Run’ in the ‘Prog’ menu, or directly from the RunOpl icon in the System screen.

Use Psion-Esc to stop a running program.

Use Control-S to pause a program and any other key to restart it.

Use Psion-Menu to display a status window.

VARIABLES & CONSTANTS

Programs can process data in a variety of ways. They may, for example, perform calculations with numbers, or save and recall *strings* of text (such as names and phone numbers in a data file).

In all cases, your program must be able to handle *values* - different types of numbers, strings, and so on.

In OPL, there are two ways of handling values: *variables* and *constants*. Constants are fixed values (which may be named on the Series 5). Variables are used to store values which may change - for example, a variable called X **may start with the value 3 but later take the value 7.**

DECLARING VARIABLES

Most procedures begin by *declaring* (creating) variables:

```
LOCAL x,y,z
```

LOCAL is the word telling the Psion to create variables, with the names which follow - here `x`, `y` and `z` — separated by commas.

The statement `LOCAL x,y,z` defines three variables called `x`, `y` and `z`. The Psion will recognise these names whenever you use them in this procedure. (If you used them in another procedure, they wouldn't be recognised; the variables are 'local' to the procedure in which they are declared.)

These variables are initially given the value 0.

Any variables you wish to use must be declared at the **start** of a procedure.

CHOOSING THE VARIABLE

Before declaring variables, decide what information they are going to contain. There are different types of variables for different sorts of values. If you try to give the wrong type of value to a variable, an error message will be displayed.

You specify the type of each variable when you declare it, by adding a symbol at the end of its name.

NUMBERS

- For small whole numbers - for example 6 - use an ***integer variable***. Integer variables have a % symbol on the end, for example `number%`.
  Integer variables can handle numbers only in the range -32768 to +32767. If you try to give an integer variable a whole number bigger than this, an error message will be displayed. If a variable may have to handle numbers outside normal integer range, make it a long integer variable.

- For larger whole numbers - for example 10000000 - use a ***long integer variable***. Long integer variables have an & symbol on the end, for example `number&`.
  Long integer variables can handle whole numbers in the range -2147483648 to +2147483647.

- For non-whole numbers - for example 2.5 - use a ***floating-point variable***. Floating-point variables have no symbol on the end: `price`, for example.
  If you know that at some stage in your program your variable will have to handle non-whole numbers, like 1.2, use a floating-point, not an integer variable. Otherwise you may get unpredictable results. (There's more about this later in this section.)

- For very large whole numbers outside long integer range you should also use floating-point variables.

❺ It is possible to use the full available range of 64-bit floating-point values, i.e. all real numbers with absolute values in the range 2.2250738585072015E-308 to 1.7976931348623157E+308 and 0. Precision remains limited to about 15 significant figures in this range. It is also possible to use numbers which have absolute values in the range 5E-324 to 2.2250738585072015E-308 (called *denormals*), however the precision decreases in this range to only 1 significant figure at the lower end. It is possible to enforce the ranges used by the Series 3c and other earlier Psion machines (see the Series 3c section below) by using the SETFLAGS command. See the 'Alphabetic Listing' section of the 'Glossary.pdf' document for details of this.

Constants for the maximum and minimum values of all variable types are given in Const.oph. See the 'Calling Procedures' section of this document for details on how to use this file and Appendix E in the 'Appends.pdf' document for a listing of it.

❸ Floating-point variables can handle numbers as big as ±9.99999999999e99 and as small as ±1e-99, and 0. Intermediate results in calculations (which are not displayed on the screen) may exceed this and take any value in the full range of 64-bit floating point numbers (see the Series 5 section above) .

TEXT

For text - `Are you sure?`, `54th`, etc. - use a ***string variable***. (Pieces of text are called *strings* in OPL.) String variables have a `$` symbol on the end - for example, `name$`.

To declare a string variable, you **must** follow the `$` symbol with the maximum length of string you want the variable to handle in brackets. So if you want to store names up to 15 characters long in the variable `name$`, declare it like this: `LOCAL name$(15)`Strings cannot be longer than 255 characters.

ARRAY VARIABLES

You may want a group of variables, for example to store lists of values. Instead of having to declare separate variables `a`, `b`, `c`, `d` and `e`, you can declare *array* variables `a(1)` to `a(5)` in one go like this:

`LOCAL a%(5)`	(array of integer variables)
`LOCAL a(5)`	(array of floating-point variables)
`LOCAL a$(5,8)`	(array of string variables)

or

`LOCAL a&(5)`	(array of long integers)

The number in brackets is the number of elements in the array. So `LOCAL a%(5)` creates five integer variables: `a%(1)`, `a%(2)`, `a%(3)`, `a%(4)` and `a%(5)`.

With strings, the second number in the brackets specifies the maximum length of the strings. All the elements in the string array have the same capacity - for example, `LOCAL ID$(5,10)` allocates memory space for five strings, each up to 10 characters in length.

OPL does not support two-dimensional arrays.

INITIAL VALUES

All numeric variables have zero as their initial value. String variables have a string with no characters in it (a *null* or *empty* string). Every element in an array variable is also initialised in the appropriate way.

CHOOSING DESCRIPTIVE NAMES

To make it easier to write your programs, and understand them when you read through them at a later date, give your main variables names which describe the values they hold. For example, in a procedure which calculates fuel efficiency, you might use variables named `speed` and `distance`.

All variable names:

- May be entered in any combination of upper and lower case. sPeeD and SpEEd would be considered the same name.
- Must not use any of the names of keywords, as listed in the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document - if you use these you will see a ‘Declaration error’ message when you translate your module.

Other constraints are machine dependent:

❺

- May be up to 32 characters long
- **Must** start with either an underscore (_) or an alphabetic character, but after that may use any combination of numbers, letters and the underscore character.

❸

- May be up to 8 characters long
- **Must** start with an alphabetic character, but after that may use any combination of numbers and letters

The `$`, `&` and `%` symbols are included in the 32 (or 8) characters allowed in variable names, so `V23456789012345678901234567890l2%` is too long to be a valid variable name, but `V2345678901234567890123456789Ol%` is acceptable (on the Series 5).

EXAMPLES

- `LOCAL clients$(12),z&(3)` declares one string variable, `clients$`, of capacity 12 characters, and one long integer array variable containing three elements, `z&(1)`, `z&(2)` and `z&(3)`
- `LOCAL AGE%,B5$(10),i` declares one integer variable, `AGE%`, one string variable, `B5$`, of capacity 10 characters, and one floating-point variable, `i`
- `LOCAL profit93` declares one floating-point variable, `profit93`
- `LOCAL x,MAN6$(4,7)` declares one floating-point variable, `x`, and one string array variable, `man6$`, containing four elements, `man6$(1)`, `man6$(2)`, `man6$(3)` and `man6$(4)`, each of capacity 7 characters

FOR EFFICIENCY

- Integer variables use less memory than long integer variables, and both use less than floating-point.
- Integer variables are processed faster than floating-point.

GIVING VALUES TO VARIABLES

ASSIGNING VALUES

You can *assign* a value to a variable directly, like this:

```
x=5
```

```
y=10
```

This procedure adds two numbers together:

```
PROC add:
     LOCAL x%,y%,z%
     x%=569
     y%=203
```

```
    z%=x%+y%
    PRINT z%
    GET
ENDP
```

`add:` is the procedure name.

The LOCAL statement defines three variables `x%`, `y%` and `z%`, all initially with the value 0. PRINT displays the value of `z%` on the screen. You can display the value of any variable like this.

PROC and ENDP define the beginning and end of the procedure as you saw in the previous section.

ASSIGNING VALUES TO STRING VARIABLES

String variables can be assigned text values like this:

```
a$=“some text”
```

The text you use must be enclosed in double quote characters.

ASSIGNING VALUES TO AN ARRAY VARIABLE

If you declare `a%(4)`, assign values to each of the elements in the array like this: `a%(1)=56`, `a%(2)=345` and so on. Similarly for the other variable types: `a(1)=.0346`, `a&(3)=355440`, `a$(10)=“name”`.

ARITHMETIC OPERATIONS

You can use these *operators*:

+	plus
-	minus or make negative
/	divide
*	multiply
**	raise to a power
%	percentage

Operators have the same precedence as in the Calc application. For example, `3+51.3/8` is treated as `3+(51.3/8)`, not `(3+51.3)/8`. For more information on operators and precedence, see Appendix B.

VALUES FROM FUNCTIONS

There are two kinds of keyword - *commands* and *functions*:

- A command is just a straightforward instruction to OPL to do some particular thing. PRINT and PAUSE, for example, are commands.
- A function is just like a command but it also *returns* a value which you can then use.

GET is, in fact, a function; it waits for you to press a key on the keyboard, and then returns a value which identifies the key which was pressed. (In previous example programs, the value returned by GET was ignored, as GET was being used to provide a pause while you read the screen. This is a common use of the GET function.)

The number returned by GET will always be a small whole number, so you might store it away in an integer variable, like this:

```
a%=GET
```

There is more about the GET function later in this section.

EXPRESSIONS

You can assign a value to a variable with an *expression* - that is, a combination of numbers, variables, and functions. For example:

`z=x+y/2`	gives the `z` the value of `x` plus the value of `y/2`.
`z=x*y+34.78`	gives `z` the value of `x` times `y`, plus `34.78`.
`z=x+COS(y)`	gives `z` the value of `x` plus the cosine of `y`.

COS is another OPL function. Unlike the GET function, COS requires a value or variable to work with. As you can see, you put this in brackets, after the function name. Values you give to functions in this way are called *arguments* to the function. There is more information about arguments in the next section.

All of the above are *operations* using the variables `x` and `y` - assigning the result to `z` and not actually affecting the value of `x` or `y`.

The ways you can change the values of variables fall into these groups:

- Arithmetic operations, such as multiplication or addition - for example `z=sales+costs` or `z=y%*(4-x%)`

- Using one of the OPL functions, for example `z=SIN(PI/6)`

or

- Using certain keywords like INPUT or EDIT which wait for you to type in values from the keyboard.

SELF REFERENCE

In expressions, variables can refer to themselves. For example:

`z%=z%+1`	(make the value of `z%` one greater than its current value)
`x%=x%/4+y`	(make the value of `x%` a quarter of its current value, plus the value of `y`)

CONSTANTS

In an OPL program, numbers (and strings in quote marks) are sometimes called *constants.* In practice, you will use constants without thinking about them. For example:

```
x=0.32

x%=569

x&=32768

x$=“string”

x(1)=4.87
```

OPL can also represent *hexadecimal* constants. Integers specified in hexadecimal must be preceded by a `$` and long integers by a &. For example, $f or &80000000. This is explained under the HEX$ entry in the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document.

Exponential notation may be useful for very large or very small numbers. Use `E` (capital or lower case) to mean “times ten to the power of” - for example, 3.14E7 is 3.14×10^{7} (31400000), while 1E-9 is 1×10^{-9} (0.000000001).

❺ The CONST command may be used to declare *constants*. This makes it possible to assign a name to a constant value so it may be used throughout the module. This has the advantage of making it possible to change just one statement rather than many to change the value of a single constant. See the ‘Calling Procedures’ section of this document for more details of how to do this.

PROBLEMS WITH INTEGERS

When calculating an expression, OPL uses the simplest arithmetic possible for the numbers involved. If all of the numbers are integers, integer arithmetic is used; if one is outside integer range, but within long integer range, then long integer arithmetic is used; if any of the numbers are not whole numbers, or are outside long integer range, floating-point arithmetic is used.

This has the benefit of maximising speed, but you must beware of calculations going out of the range of the type of arithmetic used. For example, in `X=200*300` both `200` and `300` are integers, so integer arithmetic is used for speed (even though `X` is a floating-point variable). However, the result, 60000, cannot be calculated because it is outside integer range (32767 to -32768), so an ‘Overflow’ error is produced.

You can get around this by using the INT function, which turns an integer into a long integer, without changing its value. If you rewrite the previous example as `X=INT(200)*300`, OPL has to use long integer arithmetic, and can therefore give the correct result (60000). (If you understand hexadecimal numbers, you can instead write one of the numbers as a hexadecimal long integer, e.g. `200` would become `&C8`.)

Integer arithmetic uses whole numbers only. For example, if `y%` is 7 and `x%` is 4, `y%/x%` gives 1. However, you can use the INTF function to convert an integer or long integer into a floating-point number, forcing floating-point arithmetic to be used for example, `INTF(y%)/x%` gives 1.75. **This rule applies to each part of an expression** - e.g. `1.0+2/4` works out as `1.0+0` (=1.0), while `1+2.0/4` works out as `1+0.5` (=1.5).

If one of the integers in an all-integer calculation is a constant, you can instead write it as a floating-point number. 7/4 gives 1, but 7/4.0 gives 1.75.

OPERATIONS ON STRINGS

If `a$` is `“down”` and `b$` is `“wind”`, then the statement `c$=a$+b$` means `c$` becomes `“downwind”`.

Alternatively, you could give `c$` the same value with the statement `c$=“down”+”wind”`.

When adding strings together, the result must not be longer than the maximum length you declared e.g. if you declared `LOCAL a$(5)` then `a$=“first”+”second”` would cause a ‘String is too long’ error to be displayed.

Most operators do not work on strings. To cut up strings, use string functions like MID$, LEFT$ and RIGHT$, explained in the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document. You need them to extract even a single character you **cannot**, for example, refer to the fourth character in `a$(7)` as `a$(4)`.

DISPLAYING VARIABLES

PRINT is one of the most useful OPL commands. Use it to display any combination of text messages and the values of variables.

WHERE THE CURSOR GOES AFTER A PRINT

In general, each PRINT statement ends by moving to a new line. For example:

```
A%=127 :PRINT “A% is”
PRINT a%
```

would display as

```
A% is
127
```

You can stop a PRINT statement from moving to a new line by ending it with a semicolon. For example:

```
A%=127 :PRINT “A% is”;
PRINT a%
```

would display as

```
A% is127
```

If you end a PRINT statement with a comma, it stays on the same line, but displays an extra space. For example:

```
A%=127 :PRINT “A% is”,
PRINT a%
```

would display as

```
A% is 127
```

DISPLAYING A LIST OF THINGS

You can use commas or semicolons to separate things to be displayed on one line, instead of using one PRINT statement for each. They have the same effect as before:

```
A%=127 :PRINT “A% is”,a%
```

would display as

```
A% is 127
```

while

```
user$=“Fred”
PRINT “Hello”,user$;“!”
```

would display as

```
Hello Fred!
```

DISPLAYING THE QUOTE CHARACTER

Each string you use with PRINT must start and end with a quote character. Inside the string to display, you can represent the quote character itself by entering it twice. So `PRINT “Press “” key”` displays as `Press “ key`, while `PRINT “”””` displays a single quote character.

VALUES FROM THE KEYBOARD

If you want a program to be reusable, it often needs to be able to accept different sets of information each time you use it. You can do this with the INPUT command, which takes numbers and text typed in at the keyboard and stores them in variables.

For example, this simple procedure converts from Pounds Sterling to Deutschmarks. It asks you to type in two numbers - the number of Pounds Sterling, and the current exchange rate. You can edit as you type the numbers - the Delete key, for example, deletes characters, and Esc clears everything you've typed. Press Enter when you've finished each number. The values are assigned to the variables `pounds` and `rate`, and the result of the conversion is then displayed:

```
PROC exch:
  LOCAL pounds,rate
  AT 1,4
  PRINT “How many Pounds Sterling?”,
  INPUT pounds :REM value from keyboard
  PRINT “Exchange rate (DM to £1)?”,
  INPUT rate :REM value from keyboard
  PRINT “=”,pounds*rate,“Deutschmarks”
  GET
ENDP
```

Here PRINT is used to show messages (often called *prompts*) before the two INPUT commands, to say what information needs to be typed in. In both cases the PRINT command ends in a comma, which displays a single space, and keeps the cursor position on the same line. Without the commas, the numbers you type to the INPUT commands would appear on the line below.

The value entered to an INPUT command must be of the appropriate kind for the variable which INPUT is setting. If you enter the wrong type (for example, if you enter the string `three` for the floating-point variable `rate`), INPUT will show a `?` prompt, and wait for you to enter another value.

When using INPUT with a numeric variable (integer, long integer or floating-point), you can enter any number within the range of that type of variable. Note that if you enter a non-whole number as the value for an integer variable, it will take only the whole number part (so e.g. if you enter 12.75 for an integer variable, it will be set to 12).

COMMENTS

The REM command lets you add comments to a program to help explain how it works. Begin the comment with the word REM (short for ‘remark’). Everything after the REM command is ignored.

If you put a REM command on the end of a line, the colon you would normally put before it is optional. For example, you could use either of these:

```
  CLS :REM Clears the screen
```

or

```
  CLS REM Clears the screen
```

AT COMMAND

This positions the cursor or your message at the co-ordinates you specify. Use the command like this:

```
AT column%,row%
```

where `column%` and `row%` give the character position to use.

`AT 1,1` positions the cursor to the top left corner.

SINGLE KEYPRESSES

In addition to using INPUT to ask for values, your program can ask for single keypresses. Use one of these functions:

- GET waits for a keypress and returns the key pressed.
- KEY returns a key if any was pressed, but doesn't wait for one.

Every separate letter, number or symbol has a number which represents it, called a *character code*. The full list of character codes - the *character set* - for the Series 5 may be found in Appendix D and for the Series 3c is included as an appendix to the User Guide. GET and KEY return the character code of the key pressed for example, if A were pressed, these functions would return the value 65. KEY returns 0 if no key was pressed.

KEY$ and GET$ work in the same way as KEY and GET, except that they return the key pressed as a string, not as a character code:

- GET$ waits for a keypress and returns the key pressed, as a string.
- KEY$ returns a key if any was pressed, but doesn't wait for one. KEY$ returns a null string if no key was pressed.

Unlike INPUT, these functions do not display the key pressed on the screen, and do not wait for you to press Enter.

EXAMPLE USING GET$

```
PROC kchar:
   LOCAL k$(1)
   PRINT “Press a key, A-Z:”
   k$=GET$
   PRINT “You pressed”,k$
   PAUSE 60
ENDP
```

Single keypresses are often useful for making decisions. A program might, for example, offer a set of choices which you choose from by typing the word's first letter, like this:

```
Add (A) Erase (E) or Copy (C) ?
```

Or it might ask for confirmation of a decision, by displaying a `YES or NO?` message and waiting until Y or N is pressed.

See the ‘Loops and Branches’ section of this document for details of how to identify which key is pressed.

MODIFIER KEYS

If you need to check for the Shift, Control, Psion (on the Series 3c and Siena only) Fn (Series 5 only) keys and/or Caps Lock being used, see the description of the KMOD function, in the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document.

SUMMARY

Declare variables with one or more LOCAL statements in the line after PROC:

- ***Integer*** variables - for example year%
- ***Floating-point*** variables - for example price
- ***String*** variables - for example name$(12) where the maximum length is given in the brackets
- ***Long integer*** variables - for example profit&

Variables will be floating-point unless you add a symbol to the end of the variable name.

- ***Array*** variables - for example prices%(4) or clients$(5,12) where the first number inside the brackets specifies the number of elements, and the second number in the brackets, in the case of string arrays, specifies the maximum length.

All identifiers may have a maximum length of 32 characters (8 on the Series 3c).

Assign values to variables:

- Expressions - for example x=5.5/y , profit=x-y
- INPUT command - for example INPUT a$
- ‘Add’ strings - for example a$=“MR”+names$

REM allows you to add comments to a program.

AT positions the cursor.

GET and KEY return the key pressed as a character code.

GET$ and KEY$ return the key pressed as a single-character string.

GET and GET$ wait until a key is pressed, KEY and KEY$ do not.

LOOPS & BRANCHES

The programs in the two previous sections consist of a number of instructions which are executed one by one, from start to finish.

However, there are a number of other ways a program can proceed:

- **Repeating a set of instructions (called loops)**
- **Doing one set of instructions or another (called IF statements)**
- **Jumping from one line of your program to another**

REPEATING INSTRUCTIONS (LOOPS)

The DO...UNTIL and WHILE...ENDWH commands are *structures* - they don’t actually do anything to your data, but control the order in which other commands are executed:

- DO...UNTIL repeats a set of instructions until a certain condition is true.
- WHILE...ENDWH repeats a set of instructions so long as a certain condition is true.

There is a test *condition* at the end of the DO...UNTIL loop, and at the beginning of the WHILE...ENDWH loop.

DO...UNTIL

```
PROC test:
   LOCAL a%
   a%=10
   DO
      PRINT “A=”;a%
      a%=a%-1
   UNTIL a%=0
   PRINT “Finished”
   GET
ENDP
```

The instruction `DO` says to OPL:

“Execute all the following instructions until an UNTIL is reached. If the condition following UNTIL is not met, repeat the same set of instructions until it is.”

The first time through the loop, `a%`=10. 1 is subtracted from `a%`, so that `a%` is 9 when the UNTIL statement is reached. Since `a%` isn’t zero yet, the program returns to DO and the loop is repeated.

`a%` goes down to 8, and again it fails the UNTIL condition. The loop therefore repeats 10 times until `a%` does equal zero.

When `a%` equals zero, the program continues with the instructions after UNTIL.

The statements in a DO...UNTIL loop are always executed at least once.

WHILE...ENDWH

```
PROC test2:
   LOCAL a%
   a%=10
   WHILE a%>0
      PRINT “A=”;a%
      a%=a%-1
   ENDWH
   PRINT “Finished”
   GET
ENDP
```

The instructions between the WHILE and ENDWH statements are executed only if the condition following the WHILE is true - in this case if `a%` is greater than 0.

Initially, `a%`=10 and so `A=10` is displayed on the screen. `a%` is then reduced to 9. `a%` is still greater than zero, so `A=9` is displayed. This continues until `A=1` is displayed. `a%` is then reduced to zero, and so `Finished` is displayed.

Unlike DO...UNTIL, it's possible for the instructions between WHILE and ENDWH not to be executed at all.

EXAMPLE USING WHILE...ENDWH

```
PROC newkey:
  WHILE KEY :ENDWH
  PRINT "Press a new key."
ENDP
```

This procedure ignores any keys which may already have been typed, then waits for a new keypress.

KEY returns the value of a key that was pressed, or `0` if no key has been pressed. `WHILE KEY :ENDWH` reads any keys previously pressed, one by one, until they have all been read and `KEY` returns zero.

CHOOSING BETWEEN INSTRUCTIONS

In a program, you might have several possible cases (`x%` may be 1, or it may be 2, or 3...) and want to do something different for each one (if it's 1, do this, but if it's 2, do that...). You can do this with the IF...ENDIF structure:

```
IF condition1
     do these statements
ELSEIF condition2
     do these statements
ELSEIF condition3
     do these statements
     .
     .
ELSE
     do these statements
ENDIF
```

These lines would do **either**

- the statements following the IF line (if *condition1* is met)

or

- the statements following one of the ELSEIF lines (if one of *condition2*, *condition3*... is met)

or

- the statements following the ELSE line (if none of *condition1*, *condition2*, *condition3*... have been met).

and then continue with the statements after the ENDIF.

You can cater for as many cases as you like with ELSEIF statements. You don't have to have any ELSEIFs. There may be either one ELSE statement or none; you do not specify conditions for the ELSE statement.

Every IF in your program must be matched by an ENDIF - otherwise you'll see an error message when you try to translate the module. The structure must start with an IF and end with an ENDIF.

"NESTING" LOOPS - THE 'TOO COMPLEX' MESSAGE

You can have up to eight DO...UNTIL, WHILE...ENDWH and/or IF...ENDIF structures nested within each other. If you nest them any deeper, a 'Too complex' error message will be displayed.

EXAMPLE USING IF

```
PROC zcode:
  LOCAL g%
  PRINT "Are you going to press Z?"
  g%=GET
  IF g%=%Z OR g%=%z
    PRINT "Yes!"
  ELSE
    PRINT "No."
  ENDIF
  PAUSE 60
ENDP
```

% OPERATOR

The program checks character codes with the `%` operator. `%a` returns the code of `a`, `%Z` the code of `Z` and so on. Using `%A` is entirely equivalent to using 65, the actual code for `A`, but it saves you having to look it up, and it makes your program easier to follow.

Be careful not to confuse character codes like these with integer variables.

OR OPERATOR

OR lets you check for either of two conditions. OR is an example of a *logical operator*. There is more about logical operators later in this section.

EXAMPLE USING DO...UNTIL AND IF

```
PROC testny:
  DO
  g$=UPPER$(GET$)
  UNTIL g$="N" OR g$="Y"     REM wait for a Y or N
  IF g$="N"                  REM was it an N?
    ... REM 'N' pressed
  ELSE                       REM must have been a Y
    ... REM 'Y' pressed
  ENDIF
ENDP
```

This procedure checks for a 'Y' or 'N' keypress. You'd put your own code in the IF statement, where `...` has been used.

ARGUMENTS TO FUNCTIONS

Some functions, as with commands like PRINT and PAUSE, require you to give a value or values. These values are called *arguments*. The UPPER$ function needs you to specify a string argument, and returns the same string but with all letters in upper case. For example, `UPPER("12.+aBcDeF")` returns `12.+ABCDEF`.

FUNCTIONS AS ARGUMENTS TO OTHER FUNCTIONS

Since GET$ returns a string, you can use this as the argument for UPPER$. `UPPER$(GET$)` waits for you to press a key, because of the GET$; the UPPER$ takes the string returned and, if it's a letter, returns it in upper case. This means that you can check for `Y` without having to check for `y` as well.

'TRUE' AND 'FALSE'

The test condition used with DO...UNTIL, WHILE...ENDWH and IF...ENDIF can be any expression, and may include any valid combination of operators and functions. Examples:

Condition	*Meaning*
`x=21`	does the value of `x` equal 21? (Note - as this is a test condition, it does **not** assign `x` the value 21)
`a%<>b%`	is the value of `a%` not equal to the value of `b%`?
`x%=(y%+z%)`	is the value of `x%` equal to the value of `y%+z%`? (does not assign the value `y%+z%` to `x%`).

The expressions actually return a *logical value* - that is, a value meaning either 'True' or 'False'. Any non-zero value is considered 'True' (to return a 'True' value, OPL uses -1), while zero means 'False'. So if `a%` is 6 and `b%` is 7, the expression `a%>b%` will return a zero value, since `a%` is **not** greater than `b%`.

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

These are the conditional operators:

<	less than	<=	less than or equal to
>	greater than	>=	greater than or equal to
=	equal to	<>	not equal to

LOGICAL OPERATORS

The operators AND, OR and NOT allow you to combine or change test conditions. This table shows their effects. (`c1` and `c2` represent conditions.)

Example	*Result*	*Integer returned*
`c1 AND c2`	True if both `c1` and `c2` are true	-1
	False if either `c1` or `c2` are false	0
`c1 OR c2`	True if either `c1` or `c2` is true	-1
	False if both `c1` and `c2` are false	0
`NOT c1`	True if `c1` is false	-1
	False if `c1` is true	0

However, AND, OR and NOT become *bitwise operators* - something very different from logical operators - **when used exclusively with integer or long integer values**. If you use `IF A% AND B%`, the AND acts as a bitwise operator, and you may not get the expected result. You would have to rewrite this as `IF A%<>0 AND B%<>0`. (Operators, including bitwise operators, are discussed further in Appendix B in the 'Appends.pdf' document.)

JUMPING TO A DIFFERENT LINE

JUMPING OUT OF A LOOP: BREAK

The BREAK command jumps out of a DO...UNTIL or WHILE...ENDWH structure. The line after the UNTIL or ENDWH statement is executed, and the lines following are then executed as normal. For example:

```
DO
  ...
  IF a=c
    BREAK
  ENDIF
  ...
UNTIL a=b[PC1]
x%=3
...
```

JUMPING TO THE TEST CONDITION: CONTINUE

The CONTINUE command jumps from the middle of a loop to its test condition. The test condition is either the UNTIL line of a DO...UNTIL loop or the WHILE line of a WHILE...ENDWH loop. For example:

```
WHILE a<b
  ...
  IF a=c
    CONTINUE
  ENDIF
  ...
ENDWH
...
```

JUMPING TO A ‘LABEL’: GOTO

The GOTO command jumps to a specified *label*. The label can be anywhere in the same procedure (after any LOCAL or GLOBAL variable declarations). In this example, when the program reaches the GOTO statement, it jumps to the label `exit::`, and continues with the statement after it.

```
GOTO exit
PRINT “MISS THIS LINE”
PRINT “AND THIS ONE”
exit::
```

The two PRINT statements are missed out.

Labels themselves **must** end in a double colon. This is optional in the GOTO statement - both `GOTO exit::` and `GOTO exit` are OK.

The jump to the label always happens - it is not conditional.

Don’t use GOTOs instead of DO...UNTIL or WHILE...ENDWH, as they make procedures difficult to understand.

VECTORING TO A LABEL: VECTOR/ENDV

VECTOR jumps to one of a list of labels, according to the value in an integer variable. The list is terminated by the ENDV statement. For example:

```
     VECTOR p%
          FUNCA,FUNCX
          FUNCR
     ENDV
     PRINT “p% was not 1/2/3” :GET :STOP
FUNCA::
     PRINT “p% was 1” :GET :STOP
FUNCX::
     PRINT “p% was 2” :GET :STOP
FUNCR::
     PRINT “p% was 3” :GET :STOP
```

Here, if `p%` is 1, VECTOR jumps to the label `FUNCA::`. If it is 2, it jumps to `FUNCX::`, and if 3, `FUNCR::`. If `p%` is any other value, the program continues with the statement after the `ENDV` statement.

STOPPING A PROGRAM

The above example introduces the STOP command. This stops a running program completely, just as if the end of the program had been reached. In a module with a **single** procedure, STOP has the same effect as using GOTO to jump to a label above the final ENDP.

UNTIL 0, WHILE 1

Zero and non-zero are logical values meaning ‘False’ and ‘True’ respectively. `UNTIL 0` and `WHILE 1` therefore mean ‘do forever’, since the condition 0 is never ‘True’ and the condition 1 is always ‘True’. Use loops with these conditions when you need to check the real condition somewhere in the middle of the loop. When the real condition is met, you can BREAK out of the loop.

For example:

```
PROC test:
     WHILE 1
          ... REM some other lines here
     IF KEY :BREAK :ENDIF
          ... REM some other lines here
     ENDWH
ENDP
```

This example uses the KEY command. KEY returns 0 if no key has been pressed. When a key is pressed, KEY returns a non-zero value which counts as ‘True’, and the BREAK is executed.

SUMMARY

```
DO
     statements
UNTIL condition
```

```
WHILE condition
     statements
ENDWH
```

```
IF condition
     statements
[ELSEIF condition
     statements]
[ELSE
     statements]
ENDIF
```

```
VECTOR int%
     label1, label2
     label3...
ENDV
...
label1::
...
label2::
...
label3::
...
```

`GOTO label` jumps to `label::`

`BREAK` goes to the first line after the end of the loop - the line following the UNTIL or ENDWH line.

`CONTINUE` goes to the test condition of the loop - the UNTIL or the WHILE line.

`STOP` stops a running program completely.

CALLING PROCEDURES

The programs discussed in earlier sections have involved a single procedure in each module. However, it is possible to have more than one procedure in a module.

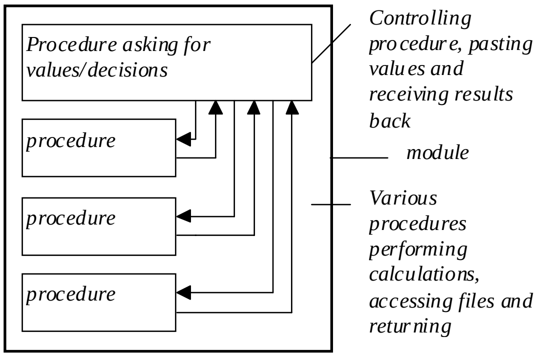
The top procedure is always the one which is executed, but it may also call, by name, any of the other procedures in the module. This procedure may in turn return a value, for example the result of a calculation, to the calling procedure.

Variables can be made available to all the procedures in a module by using the GLOBAL rather than LOCAL definition.

USING MORE THAN ONE PROCEDURE

If you wanted a single procedure to perform a complex task, the procedure would become long and complicated. It is more convenient to have a module containing a number of procedures, each of which you can write and edit separately.

Many OPL modules are in fact a set of procedures linked up - each procedure doing just one job (such as a certain calculation) and then passing its results on to other procedures, so they can do other operations:


Procedure asking for values/decisions
procedure
procedure
procedure
Controlling procedure, pasting values and receiving results back
module
Various procedures performing calculations, accessing files and returning

OPL is designed to encourage programs written in this way, since:

- You can store all the procedures which make up a program in the same module file

and

- One procedure can *call*, that is run, another.

MODULES CONTAINING MORE THAN ONE PROCEDURE

You can have as many procedures as you like in a module. Each must begin with PROC and end with ENDP.

When you run a translated module it is always the first procedure, at the top of the module, which is actually run. When this finishes, the module stops; any other procedures in the file are only run if and when they are called.

Although you can use any name you want, it’s common to give the first procedure a name like `start`.

Procedures which run on their own should be written and translated as separate modules, otherwise you won’t be able to run them.

CALLING PROCEDURES

To run another procedure, simply give the name of the procedure (with the colon). For example, this module contains two procedures:

```
PROC one:
   PRINT “Start”
   PAUSE 40
   two:                    REM calls procedure two:
   PRINT “Finished”
   PAUSE 40
ENDP
```

```
PROC two:
  PRINT “Doing...”
  PAUSE 40
ENDP
```

Running this module would run procedure `one:`, with this effect: `Start` is displayed; after a PAUSE it calls `two:`, which displays `Doing...`; after another PAUSE `two:` returns to the `one:` procedure; `one:` displays `Finished`; and after a final PAUSE, `one:` finishes.

❺ Remember the ‘Go to’ button on the toolbar allows you to jump between procedures, for quick navigation around the module.

❸ Remember the diamond key allows you to switch between a ‘Normal’ and an ‘Outline’ view of your OPL module. The ‘Outline’ view lists only the names of each procedure, for quick navigation around the module.

USES OF CALLING PROCEDURES

Calling procedures can be used to:

- Structure your programs more clearly so they’re easier to adapt after you’ve written them
- Use the same procedure in different programs - say, to perform a certain common calculation.

For example, when your program asks you “Do this or do that?”, make two procedure calls - either `this:` or `that:` procedure - depending on what you reply, for example:

```
PROC input:
  LOCAL a$(1)
  PRINT “Add [A] or Subtract [S]?:”,
  a$=UPPER$(GET$)
  IF a$=“A”
    add:                     REM first procedure
  ELSEIF a$=“S”
    subtract:                REM second procedure
  ENDIF
ENDP
```

To make full use of procedure calls, you must be able to communicate values between one procedure and another. There are two ways of doing this: *global variables* and *parameters*.

PARAMETERS

Values can be passed from one procedure to another by using *parameters*. They look and act very much like arguments to functions.

In the example below, the procedure `price:` calls the procedure `tax:`. At the same time as it calls it, it passes a value (in this case, the value which INPUT gave to the variable `x`) to the parameter `p` named in the first line of `tax:`. The parameter `p` is rather like a new local variable inside `tax:`, and it has the value passed when `tax:` is called. (The `tax:` procedure is **not** changing the variable `x`.)

The `tax:` procedure displays the value of `x` plus 17.5% tax.

```
PROC price:
  LOCAL x
  PRINT “ENTER PRICE”,
  INPUT x
  tax:(x)         REM Passes the value of x to p
  GET
ENDP


PROC tax:(p)
  PRINT “PRICE INCLUDING TAX =”,p*1.175
ENDP
```

- In the *called* procedure, follow the procedure name by the names to be used for the parameters, enclosed by brackets and separated by commas - for example proc2:(cost,profit).

The parameter type is specified as with variables - for example `p` for a floating-point parameter, `p%` for an integer, `p&` for a long integer, `p$` for a string. You can’t have array parameters.

- In the *calling* procedure, the *values* for the parameters are given in brackets, in the right order and separated by commas, after the colon of the called procedure - for example proc2:(60,30).

The values passed as parameters may be the values of variables, strings in quotes, or constants. So a call might be `calc:(a$,x%,15.8)` and the first line of the called procedure `PROC calc:(name$,age%,salary)`

In the called procedure, you cannot assign values to parameters - for example, if `p` is a parameter, you cannot use a statement like `p=10`.

You will see a ‘Type mismatch’ error displayed if you try to pass the wrong type of value to a parameter - for example, `45` to `(a$)`.

MULTIPLE PARAMETERS

In the following example, the second procedure `tax2:` has two parameters:

- The value of the price variable x is passed to the parameter p1.

- The value of the tax rate variable r is passed to the parameter p2.

`tax2:` displays the price plus tax at the rate specified.

```
PROC price2:
  LOCAL x,r
  PRINT “ENTER PRICE”,
  INPUT x
  PRINT “ENTER TAX RATE”,
  INPUT r
  tax2:(x,r)
  GET
ENDP


PROC tax2:(p1,p2)
  PRINT p1+p2 %
ENDP
```

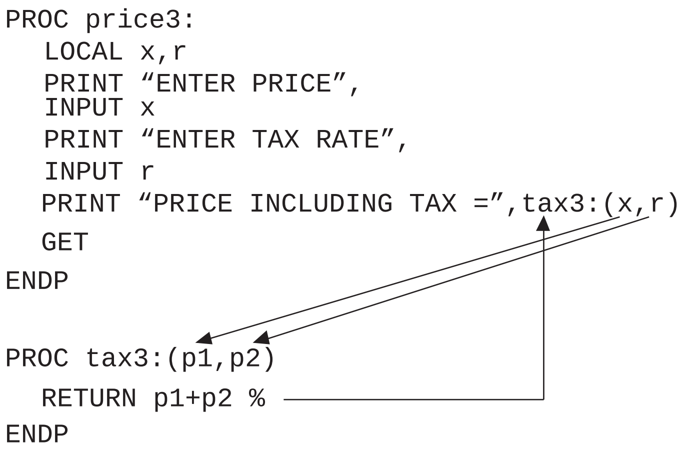
This uses the `%` symbol as an operator - `p1+p2 %` means `p1` plus `p2` percent of `p1`. Note the space before the `%`; without it, `p2%` would be taken as representing an integer variable.

See Appendix B in the ‘Appends.pdf’ document for more information about the `%` operator.

RETURNING VALUES

In the following example, the RETURN command is used to return the value of `x` plus tax at `r` percent, to be displayed in `price3:`. This is very similar to the way functions return a value.

The `tax3:` procedure calculates, but doesn't display the result. This means it can be called by other procedures which need to perform this calculation but do not necessarily need to display it.



```
PROC price3:
   LOCAL x,r
   PRINT "ENTER PRICE",
   INPUT x
   PRINT "ENTER TAX RATE",
   INPUT r
   PRINT "PRICE INCLUDING TAX =",tax3:(x,r)
   GET
ENDP

PROC tax3:(p1,p2)
   RETURN p1+p2 %
ENDP
```

Only one value may be returned by the RETURN command.

The name of a procedure which returns a value must end with the correct identifier - `$` for string, `%` for integer, or & for long integer. To return a floating-point number, it should end with none of these symbols. For example, `PROC abcd$:` can return a string, while `PROC counter%:` can return an integer. In this example, `ref$:` returns a string:

```
PROC refname:
   LOCAL a$(30),b$(2)
   PRINT "Enter reference and name:",
   INPUT a$
   b$=ref$:(a$)
   PRINT "Ref is:",b$
   GET
ENDP

PROC ref$:(name$)
   RETURN LEFT$(name$,2)
   REM LEFT$ takes first 2 letters of name$
ENDP
```

If you don't use the RETURN command, a string procedure returns the null string (`""`). Other (numeric) types of procedure return zero.

GLOBAL VARIABLES

You can only return one value with the RETURN command. If you need to pass back more than one value, use *global* variables.

Instead of declaring `LOCAL x%,name$(5)` declare `GLOBAL x%,name$(5).` The difference is that:

- Local variables are valid only in the procedure in which they are declared.
- Global variables can also be used in any procedures (including those in loaded modules) called by the procedure in which they are declared.

So this module would run OK:

```
PROC one:
   GLOBAL a%
   PRINT a%
   two:
   GET
ENDP

PROC two:
   a%=2                     REM Sees a% declared in one:
   PRINT a%
ENDP
```

When you run this, the value 0 is displayed first, and then the value 2.

You would see an ‘Undefined externals’ error displayed if you used LOCAL instead of GLOBAL to declare `a%`, since the procedure `two:` wouldn’t recognise the variable `a%`. In general, though, it is good practice to use the LOCAL command unless you really need to use GLOBAL.

A local declaration overrides a global declaration in that procedure. So if `GLOBAL a%` was declared in a procedure, which called another procedure in which `LOCAL a%` was declared, any modifications to the value of `a%` in this procedure would not effect the value of the global variable `a%`.

PASSING BACK VALUES

You can effectively pass as many values as you like back from one procedure to another by using global variables. **Any modifications to the value of a variable in a called procedure are automatically registered in the calling procedure**. For example:

```
PROC start:
   GLOBAL varone,vartwo
   varone=2.5
   vartwo=2
   op:
   PRINT varone,vartwo
   GET
ENDP

PROC op:
   varone=varone*2
   vartwo=vartwo*4
ENDP
```

This would display `5 8`

‘UNDEFINED EXTERNALS’ ERROR

If, perhaps because of a typing error, you use a name which is not one of your variables, no error occurs when you translate the module. This is because it could be the name of a global variable, declared in a different procedure, which might be available when the procedure in question was called. If no such global variable is available, an ‘Undefined externals’ error is shown when the translated module is run. This also displays the variable name which caused the error, together with the module and procedure names, in this format: ‘Error in *MODULE\PROCEDURE*, *VARIABLE*’.

SERIES 5 HEADER FILES, CONSTANTS AND PROCEDURE PROTOTYPES

On the Series 5, OPL allows you to *include header files* which may include definitions of *procedure prototypes* and *constants*, but **not** procedures themselves. (Constants and procedure prototypes may also be declared at the top of modules themselves, although it is tidier to put them into a header file. Indeed, including a file is logically identical to replacing the INCLUDE statement by the file’s contents.)

A header file is included in a module using the INCLUDE command at the beginning of the module, outside any procedure. For example,

```
INCLUDE “Header.oph”
```

The filename of the header may or may not include a path. If it does include a path, then OPL will only scan the specified folder for the file. However, the default path for INCLUDE is `\System\Opl\`, so when INCLUDE is called *without specifying a path*, OPL looks for the file firstly in the current folder and then in `\System\Opl\` in all drives from Y: to A: and then in Z:, excluding any remote drives.

Commonly the statement,

```
DECLARE EXTERNAL
```

will follow the INCLUDE declaration. DECLARE EXTERNAL causes the **translator** to report ‘Undefined externals’ errors if any variables or procedures are used before they are declared, rather than leaving this until runtime.

Procedure prototypes are declared with the command EXTERNAL. For example,

```
EXTERNAL Proc1:
```

A prototype is a declaration of the name of the procedure along with the arguments it takes. This amounts to the same as PROC declaration with the PROC keyword, which declares the start of a procedure, omitted. The procedure may then be referred to before it is defined when the DECLARE EXTERNAL statement has been made. As well as reporting ‘Undefined externals’ error at translate-time, the other advantage of using the DECLARE EXTERNAL and EXTERNAL statements is that it allows parameter type-checking to be performed at translate-time rather than at runtime, and also provides the necessary information for the translator to *coerce* numeric argument types, thus avoiding ‘Type violation’ errors at runtime. Hence a ‘Type violation’ error does not result in the following example, even though a & does not precede the 2 passed to the procedure `two:()`,

```
DECLARE EXTERNAL
EXTERNAL two:(long&)
PROC one:
     two:(2)
ENDP

PROC two:(long&)
     ..
ENDP
```

The same *coercion* occurs as when calling the built-in keywords.

Constants are declared with the command CONST. For example,

```
CONST KConstant=1.0
```

Constants are treated as literals, not stored as data. They also have *global scope* and once a value is assigned to them, it cannot be altered within the same program. The declarations must be made **outside** any procedure. A constant’s name, just like that of a GLOBAL or LOCAL variable, has the normal type-specification indicators (`%`, `&`, `$` or nothing for floats). By convention, all constants are named with a leading `K` to distinguish them from variables.

Const.oph is the standard header file in the ROM. It provides many of the standard constant declarations required for effective and maintainable OPL programming on the Series 5. For convenient reference, the contents Const.oph is reproduced in full in Appendix E. This and other files stored in the ROM (for example, OPX header files: see the ‘OPX.pdf’ document) may be created in RAM by using the ‘Create standard files’ option in the ‘Tools’ menu in the Program editor.

See also the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document.

SUMMARY

Call a procedure by stating its name, including the colon.

Pass *parameters* to a procedure by following the procedure call with the values for the parameters, e.g. `calc2:(4.5,32)`. In the called procedure, follow the procedure name with the parameter names, e.g. `PROC calc2:(mod,div%)`.

To make variables declared in one procedure accessible to called procedures, declare the variables with GLOBAL instead of LOCAL.

❺ INCLUDE may be used to *include* a *header file* which contains constant definitions and procedure prototypes.

DECLARE EXTERNAL may be used to

- cause the translator to report an error if any variables or procedures are used before they are declared
- allow parameter type-checking to be performed at translate-time rather than at runtime
- provide the necessary information for the translator to coerce numeric argument types.

Procedure prototypes are made with the EXTERNAL command.

Constant definitions are made with the CONST command.

INDEX

OPL

DATA FILE AND DATABASE HANDLING

CONTENTS

DATA FILE HANDLING

You can use OPL to create data files (databases) like those used by the Data application. You can store any kind of information in a data file, and retrieve it for display, editing or calculations.

This section covers:

- Creating data files
- Adding and editing records
- Searching records
- Using a data file both in OPL and in the Data application

The Series 5 and the Series 3c database models differ quite substantially. However, the Series 3c method of database programming (except for some removed keywords as indicated) is completely understood by the Series 5 model and any existing code will not have to change. However, it is very strongly recommended that on the Series 5 you use the new keywords INSERT, MODIFY, PUT and CANCEL, along with bookmarks and transactions, rather than using APPEND, UPDATE, POS and POSITION.

If you are using the Series 5, it is recommended that you read this section for a description of simple database use, and then the following section of this document which refers specifically to features available on the Series 5.

FILES, RECORDS AND FIELDS

Data files (or *databases*) are made up of *records* which contain data in each of their *fields*. For example, in a database of names and addresses, each record might have a name field, a telephone number field, and separate fields for each line of the address.

In OPL you can:

- Create a new ***file*** with CREATE, or open an existing file with OPEN, and copy, delete and rename files with COPY, DELETE and RENAME.
- Add a new ***record*** with APPEND, change an existing one with UPDATE, and remove a record with ERASE.
- Fill in a ***field*** by assigning a value to a field variable.

CREATING A DATA FILE

Use the CREATE command like this:

```
CREATE filename$,logical name,field1,field2,...
```

For example:

```
CREATE “clients”,B,nm$,tel$,ad1$,ad2$,ad3$
```

creates a data file called `clients`.

The file name is a string, so remember to put quote marks around it. You can also assign the name string to a string variable (for example `fil$=“clients”`) and then use the variable name as the argument - `CREATE fil$,A,field1,field2`.

LOGICAL NAMES

❺ You can have up to 26 data files open at a time. Each of these must have a logical name: `A` to `Z`.

❸ You can have up to 4 data files open at a time. Each of these must have a logical name: `A`, `B`, `C` or `D`.

The logical name lets you refer to this file without having to keep using the full file name.

A different logical name must be used for each data file opened - e.g. one called `A`, one called `B` and one called `C`. A file does not have to be opened with the same logical name as the last time it was opened. When a file is closed, its logical name is freed for use by another file.

FIELDS

`field1`, `field2`,... are the field names - up to 32 in any record. These are like variables, so - use `%` & or `$` to make the appropriate types of fields for your data. You cannot use arrays. Do not specify the maximum length of strings that the string fields can handle. The length is automatically set at 255 characters.

Field names may be up to 8 characters long, including any qualifier like &.

When referring to fields, add the logical file name to the front of the field name, to specify which opened file the fields belong to. Separate the two by a dot. For example, `A.name$` is the `name$` field of the file with logical name `A`, and `C.age%` is the `age%` field of the file with logical name `C`.

The values of all the fields are 0 or null to start with. You can see this if you run this example program:

```
PROC creatfil:
     CREATE “example”,A,int%,long&,float,str$
     PRINT “integer=”;a.int%
     PRINT “long=”;a.long&
     PRINT “float=”;a.float
     PRINT “string=”;a.str$
     CLOSE
     GET
ENDP
```

OPENING A FILE

When you first CREATE a data file it is automatically open, but it closes again when the program ends. **If a file already exists, trying to CREATE it again will give an error** - so if you ran the procedure `creatfil:` a second time you would get an error. To open an existing file, use the OPEN command.

OPEN works in the same way as the CREATE command. For example:

```
OPEN “clients”,B,nm$,tel$,ad1$,ad2$,ad3$
```

- You must use the same filename as when you first created it.

- You must include in the OPEN command each of the fields you intend to alter or read. You can omit fields from the end of the list; you **cannot** miss one out from the middle of the list, for example nm$,,ad1$ would generate an error, whereas nm$,tel$,ad1$ would be fine. They must remain the same type of field, but you can change their names. So a file created with fields name$,age% could later be opened with the fields a$,x%.

- You must give the file a logical name. See ‘Logical names’ above. You can’t have two files open simultaneously with the same logical name, so when opening the files, remember which logical names you have already used.

You might make a **new** module, and type these two procedures into it:

```
PROC openfile:
  IF NOT EXIST(“example”)
    CREATE “example”,A,int%,lng&,fp,str$
  ELSE
    OPEN “example”,A,int%,lng&,fp,str$
  ENDIF
  PRINT “Current values:”
  show:
  PRINT “Assigning values”
  A.int%=1
  A.lng&=&2**20             REM the 1st & avoids integer overflow
  A.fp=SIN(PI/6)
  PRINT “Give a value for the string:”
  INPUT A.str$
  PRINT “New values:”
  show:
ENDP
```

```
PROC show:
   PRINT “integer=”;A.int%
   PRINT “long=”;A.lng&
   PRINT “float=”;A.fp
   PRINT “string=”;A.str$
   GET
ENDP
```

NOTES

OPENING/CREATING THE FILE

The IF...ENDIF checks to see if the file already exists, using the EXIST function. If it does, the file is opened; if it doesn’t, the file is created.

GIVING VALUES TO THE FIELDS

The fields can be assigned values just like variables. The field name must be used with the logical file name like this: `A.f%=1` or `INPUT A.f$`.

If you try to give the wrong type of value to a field (for example “`Davis`” to `f%`) an error message will be displayed.

You can access the fields from other procedures, just like global variables. Here the called procedure `show:` displays the values of the fields.

FIELD NAMES

You must know the type of each field, and you must give each a separate name - you cannot refer to the fields in any indexed way, e.g. as an array.

OPENING A FILE FOR SHARING

The OPENR command works in exactly the same way as OPEN, except that the file cannot be written to (with UPDATE or APPEND), only read. However, more than one running program can then look at the file at the same time.

SAVING RECORDS

The last example procedure did not actually save the field values as a record to a file. To do this you need to use the APPEND command. This program, for example, allows you to add records to the `example` data file:

```
PROC count:
     LOCAL reply%
     OPEN “example”,A,f%,f&,f,f$
     DO
          CLS
          AT 20,1 :PRINT “Record count=”;COUNT
          AT 9,5 :PRINT “(A)dd a record”
          AT 9,7 :PRINT “(Q)uit”
          reply%=GET
          IF reply%=%q OR reply%=%Q
               BREAK
          ELSEIF reply%=%A OR reply%=%a
```

```
                        add:
                ELSE
                        BEEP 16,250
                ENDIF
        UNTIL 0
ENDP

PROC add:
        CLS
        PRINT “Enter integer field:”;
        INPUT A.f%
        PRINT “Enter long integer field:”;
        INPUT A.f&
        PRINT “Enter numeric field:”;
        INPUT A.f
        PRINT “Enter string field:”;
        INPUT A.f$
        APPEND
ENDP
```

BEEP

The BEEP command makes a beep of varying pitch and length:

```
BEEP duration%,pitch%
```

The duration is measured in 1/32 of a second, so `duration%=32` would give a beep a second long. Try `pitch%=50` for a high beep, or `500` for a low beep.

THE NUMBER OF RECORDS

The COUNT function returns the number of records in the file. If you use it just after creating a database, it will return 0. As you add records the count increases.

HOW THE VALUES ARE SAVED

Use the APPEND command to save a new record. This has no arguments. The values assigned to `A.f%`, `A.f&`, `A.f` and `A.f$` are added as a new record to the end of the `example` data file. If you only give values to some of the fields, not all, you won’t see any error message. If the fields happen to have values, these will be used; otherwise - null strings (`“”`) will be given to string fields, and zero to numeric fields.

New field values are always added to the end of the current data file - as the last record in the file (if the file is a new one, it will also be the first record).

At any time while a data file is open, the field names currently in use can be used like any other variable - for example, in a PRINT statement, or a string or numeric expression.

APPEND AND UPDATE

APPEND adds the current field values to the end of the file as a new record, whereas UPDATE deletes the **current** record and adds the current field values to the end of the file as a new record.

MOVING FROM RECORD TO RECORD

When you open or create a file, the first record in the file is current. To read, edit, or erase another record, you must make that record current - that is, move to it. Only one record is current at a time. To change the current record, use one of these commands:

- POSITION ‘moves to’ a particular record, setting the field variables to the values in that record. For example, the instruction POSITION 3 makes record 3 the current record. The first record is record 1.

- You can find the current record number by using the POS function, which returns the number of the current record.

- FIRST moves to the first record in a file.

- NEXT moves to the following record in a file. If the end of the file is passed, NEXT does not report an error, but the current record is a new, empty record. This case can be tested for with the EOF function.

- BACK moves to the previous record in the file. If the current record is the first record in the file then that first record stays current.

- LAST moves to the last record in the file.

DELETING A RECORD

ERASE deletes the current record in the current file.

The next record is then current. If the erased record was the last record in a file, then following this command the current record will be empty and EOF will return true.

FINDING A RECORD

FIND makes current the next record which has a field matching your search string. Capitals and lower-case letters match. For example:

```
r%=FIND(“Brown”)
```

would select the first record containing a string field with the value “Brown”, “brown” or “BROWN”, etc. The number of that record is returned, in this case to the variable `r%`. If the number returned is zero, no matching field was found. Any other number means that a match was found.

The search includes the current record. So after finding a matching record, you need to use NEXT before you can continue searching through the following records.

`FIND(“Brown”)` would not find a field “Mr Brown”. To find this, use wildcards, as explained below.

You can only search string fields, not number fields. For example, if you assigned the value 71 to the field `a%`, you could not find this with FIND. But if you assigned the value “71” to `a$`, you could find this.

WILDCARDS

`r%=FIND(“*Brown*”)` would make current the next record containing a string field in which `Brown` occurred - for example, the fields “MR BROWN”, “Brown A.R.” and “Browns Plumbing” would be matched. The wildcards you can use are:

?	matches any one character
*	matches any number of characters.

Once you’ve found a matching record, you might display it on the screen, erase it or edit it. For example, to display all the records containing “BROWN”:

```
FIRST
WHILE FIND(“*BROWN*”)
     PRINT a.name$,a.phone$
     NEXT
     GET
ENDWH
```

MORE CONTROLLED FINDING

FINDFIELD, like FIND, finds a string, makes the record with this string the current record, and returns the number of this record. However you can also use it to do case-dependent searching, to search backwards through the file, to search from the first record (forwards) or from the last record (backwards), and to search in one or more fields.

```
f%=FINDFIELD(a$,start%,no%,flag%)
```

searches for the string `a$` in `no%` fields in each record, starting at the field with number `start%` (1 is the number of the first field). `start%` and `no%` may refer to string fields only and other types will be ignored. The `flag%` argument specifies the type of search as explained below. If you want to search in all fields, use 1 as the second argument and for the third argument use the number of fields you used in the OPEN/CREATE command.

`flag%` should be specified as follows:

search direction	`flag%`
backwards from current record	0
forwards from current record	1
backwards from end of file	2
forwards from start of file	3

❺ 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 see Appendix E in the ‘Appends.pdf’ document for a listing of it.

Add 16 to the value of `flag%` given above to make the search *case-dependent*, where case-dependent means that the record will exactly match the search string in case as well as characters. Otherwise the search will be *case-independent* which means that upper case and lower case characters will match.

For example, if the following OPEN (or CREATE) statement had been used:

```
OPEN “clients”,B,nm$,tel$,ad1$,ad2$,ad3$
```

then the command

```
r%=FINDFIELD(“*Brown*”,1,3,16)
```

will search the `nm$`, `tel$` and `ad1$` fields of each record for strings containing “Brown” searching case-dependently backwards from the current record.

If you find a matching record and then you want to search again from this record, you must first use NEXT or BACK (according to the direction in which you are searching) to move past the record you have just found, otherwise the search will find the same match in the current record again.

CHANGING/CLOSING THE CURRENT FILE

Immediately after a file has been created or opened, it is automatically current. This means that the APPEND or UPDATE commands save records to this file, and the record-position commands (explained below) move around this file. You can still use the fields of other open files, for example `A.field1=B.field2`

USE makes current one of the other opened files. For example `USE B` selects the file with the logical name B (as specified in the OPEN or CREATE command which opened it).

If you attempt to USE a file which has not yet been opened or created, an error is reported.

In this procedure, the EOF function checks whether you are at the end of the current data file — that is, whether you've gone **past** the last record. You can use EOF in the test condition of a loop UNTIL EOF or WHILE NOT EOF in order to carry out a set of actions on all the records in a file.

EXAMPLE - COPIES SELECTED RECORDS FROM ONE FILE TO ANOTHER

```
PROC copyrec:
     OPEN “example”,A,f%,f&,f,f$
     TRAP DELETE “temp”      REM If file doesn’t exist, ignore error
     CREATE “temp”,B,f%,f&,f,f$
     PRINT “Copying EXAMPLE to TEMP”
     USE A                   REM the EXAMPLE file
     DO
          IF a.f%>30 and a.f<3.1415
          b.f%=a.f%
          b.f&=a.f&
          b.f=a.f
          b.f$=“Selective copy”
          USE B              REM the TEMP file
          APPEND
          USE A
          ENDIF
          NEXT
     UNTIL EOF               REM until End Of File
     CLOSE                   REM closes A; B becomes current
     CLOSE                   REM closes B
ENDP
```

This example uses the DELETE command to delete any `temp` file which may exist, before making it afresh. Normally, if there were no `temp` file and you tried to delete it, an error would be generated. However, this example uses TRAP with the DELETE command. TRAP followed by a command means “if an error occurs in the command, carry on regardless”. The error value can then be found using ERR.

There are more details of ERR and TRAP in the ‘Errors.pdf’ document.

CLOSING A DATA FILE

You should always ‘close’ a data file (with the CLOSE command) when you have finished using it. Data files close automatically when programs end.

❺ You can use up to 26 logical names (files or *views* — see the ‘Series 5 Database Handling’ section of this document) at a time - if you are using 26 logical names and you want to use another one, you must close one of the open files or views first. CLOSE closes the file or view referred to by the *current* logical name.

❸ You can only have 4 files open at a time - if you already have 4 files open and you want to access another one, you must close one of the open files first. CLOSE closes the *current* file.

KEEPING DATA FILES COMPRESSED

When you change or delete records in a data file, the space taken by the old information is not automatically recovered.

❺ **By default**, the space is **not** recovered when you close the file, unless you have used the SETFLAGS command to enable auto-compaction on closing a file.

❸ **By default**, the space is recovered when you close the file, provided it is on ‘Internal drive’ or on a **RAM** SSD (i.e. it is not on a Flash SSD).

Closing a very large file which contains changed or deleted records can be slow when compression is enabled, as the whole file beyond each old record needs copying down, each time.

❸ You can **prevent** data file compression on the Series 3c if you wish, with these two lines:

```
p%=PEEKW($1c)+$1e
POKEW p%,PEEKW(p%) or 1
```

(Use any suitable integer variable for `p%`.) Files used by the current program will now **not** compress when they close.

Use these two lines to re-enable auto-compression:

```
p%=PEEKW($1c)+$1e
POKEW p%,PEEKW(p%) and $fffe
```

Warning: be careful to enter these lines exactly as shown. These examples work by setting a system configuration flag.

If you have closed a file **without** compression, you can recover the space by using the COMPRESS command to create a new, compressed version of the file. `COMPRESS “dat” “new”`, for example, creates a file called `new` which is a compressed version of `dat`, with the space which was taken up by old information now recovered. (You have to use COMPRESS to compress data files which are kept on a Flash SSD.)

❺ On the Series 5, you can use the COMPACT command when the database is closed. See the ‘Series 5 Database Handling’ section of this document.

SERIES 3C AND SIENA DATA FILES AND THE DATA APPLICATION

The files you use with the Data application (listed under the Data icon in the System screen) often called *databases* or *database files* - are also just data files.

Data files created by the Data application can be viewed in OPL, and vice versa.

In OPL: to open a data file made by the Data application, begin its name with `\DAT\`, and end it with `.DBF`. For example, to open the file called `data` which the Data application normally uses:

```
OPEN “\dat\data.dbf”,A,a$,b$,c$,d$...
```

Restrictions:

- You can use up to 32 field variables, all strings. It is possible for records to contain more than 32 fields, but these fields cannot be accessed by OPL. It's safe to change such a record and use UPDATE, though, as the extra fields will remain unchanged.
- The maximum record length in OPL is 1022 characters. You will see a 'Record too large' error (-43) if your program tries to open a file which contains a record longer than this.
- The Data application breaks up long records (over 255 characters) when storing them. They would appear as separate records to OPL.

In the Data application: to examine an OPL data file, press the Data button, select 'Open file' from the 'File' menu and Control+Tab to type in a file name, and then type the name with `\OPD\` on the front and `.ODB` at the end for example:

```
\opd\example.odb
```

Restrictions:

- All of the fields must be string fields.
- You can have up to a maximum of 32 fields, as specified in the CREATE command. If you view an OPL data file with the Data application, and add more lines to records than the number of fields specified in the original CREATE command, you will get an error if you subsequently try to access these additional fields in OPL.

In both cases, you are using a more complete *file specification*. There is more about file specifications in the 'Advanced.pdf' document.

➎ For details of using Data application files in OPL on the Series 5, see the next section of this document.

SERIES 5 DATABASE HANDLING

The Series 5 uses the relational database management system (DBMS) of EPOC32 which supports SQL (Standard Query Language).

Apart from the removed keywords RECSIZE, COMPRESS and ODBINFO, the Series 3c methods of database programming are completely understood by the Series 5 model and existing code will not have to change. However, it is very strongly recommended that you use INSERT, MODIFY, PUT and CANCEL along with bookmarks and transactions, rather than using APPEND, UPDATE, POS and POSITION.

See also the 'Alphabetic Listing' section of the 'Glossary.pdf' document for some more detailed description of the use of new and changed database commands and 'Database OPX' in the 'OPX.pdf' document.

THE SERIES 5 DATABASE MODEL

As has been stressed previously, it is very strongly recommended that you use this Series 5 specific model on the Series 5, despite the fact that the data file handling methods of the Series 3c may still be used on the Series 5. The reasons for this are as follows:

- the new keywords closely reflect the underlying EPOC32 database model supplied by DBMS. They are therefore more efficient than the Series 3c keywords on the Series 5.

- to emulate the Series 3c behaviour, APPEND has to create an intermediate copy of the record which is erased on completion of the keyword. This ensures the rather strange requirement that the field values of the previous APPEND are used as the initial values for the current APPEND. This can make a database grow far larger than on the Series 3c. You can, however, use COMPACT or SETFLAGS to remove erased records from a database.

- without transactions, writing a large number of records to a database is far slower on the Series 5 than on the Series 3c. However, with transactions it is far faster.

- the Series 5 model is superior.

DATABASES, TABLES, VIEWS, FIELDS AND FIELD HANDLES

To describe the new model it is necessary to expand upon the terminology that was used in the previous section.

A Series 3c data file corresponds more or less to a single *table* in a DBMS database file. A database can contain one or more tables. A table, like a data file on the Series 3c, contains records which are made up of fields. Unlike the Series 3c, however, the field names as well as the table names are stored in the database.

CREATING DATABASES AND TABLES

With the statement:

```
CREATE “datafile”,A,f1%,f2%
```

as described in the previous section, the Series 5 creates a database called `datafile` and a table with the default name `Table1` would be added to it. The field names are derived from the `f1%` and `f2%` which are called *field handles*. The type of the field, as always, is defined by these handles.

With the Series 5 it is also possible to use, for example,

```
CREATE “people FIELDS name, number TO phoneBook”,A,n$,number$
```

This will create a table called `phoneBook` in the database called `people`, creating the database too if it does not exist. The table will have fields `name` and `number`, whose respective types are specified by the field handles `n$` and `number$`, both strings in this example.

Note that CREATE creates a **table**. An error is raised if the table already exists in the database. DBMS does not allow the database to be open when a table (or an index: see ‘Database OPX’ in the ‘OPX.pdf’ document) is created in it, so you should first close the database, i.e. close any tables previously opened in it, before using CREATE.

LOGICAL NAMES

You can have up to 26 views on tables open at a time on the Series 5. Each of these must have a logical name: `A` to `Z` (the Series 3c only supported 4 files open at one time).

FIELDS

On the Series 5, field names may be up to 32 characters long, including any qualifier like &.

OPENING DATABASES AND TABLES

With the Series 3c OPEN statement,

```
OPEN “datafile”,A,f1%,f2%
```

the Series 5 would open the default table `Table1` and provide access to as many fields as there are handles supplied.

On the Series 5, it is also possible to open multiple *views* on a table simultaneously and to specify which fields are to be available in a view, e.g.

```
OPEN “people SELECT name FROM phoneBook”,A,n$
```

This view gives you access to just the `name` field from the `phoneBook` table.

The string from `SELECT` onwards in the OPEN statement forms an SQL query which is passed straight on to the underlying EPOC32 DBMS. The SQL command-set is specified in Appendix F in the ‘Appends.pdf’ document.

A more advanced view, ordered by an *index* (described later), would be opened as follows,

```
OPEN “people SELECT name,number FROM phoneBook ORDER BY name ASC, number
DESC”,A,n$,num%
```

This would open a view with `name` fields in ascending alphabetical order and if any names were the same then the number field would be used to order these records in descending numerical order.

TRANSACTIONS

A set of related records should be committed only on successfully PUTting the last one. Otherwise all new records may be discarded using ROLLBACK. This ensures the atomicity of the whole transaction.

Transactions allow changes to a database to be *committed* in stages. It is necessary to use transactions in database operations to achieve reasonable speeds.

Transactions are a truly fundamental part of the DBMS model, so much so that without the use of transactions you will find that writing to a DBMS database is in fact slower than the equivalent operations in on the Series 3c. With transactions however, the Series 5 database handling is far faster than that of the Series 3c.

A transaction is carried out using the following commands:

- BEGINTRANS begins a transaction on the current database. Once a transaction has been started on a view (or table) then all database keywords will function as usual, but the changes to that view will not be made until COMMITTRANS is used.
- COMMITTRANS commits the transaction of the current view.
- ROLLBACK cancels the current transaction on the current view. Changes made to the database with respect to this particular view since BEGINTRANS was called will be discarded.
- INTRANS finds out whether the current view is in a transaction.

RECORD POSITION

In the DBMS model, as with most modern relational database models, absolute record position does not have much significance.

Bookmarks can be assigned to particular records to provide fast record access **and should be used in preference to POS and POSITION** when opening views using the Series 5 `OPEN...SELECT..` or `CREATE ... FIELDS...` statements. POS and POSITION can be used safely on tables opened or created using a Series 3c-style OPEN or CREATE statement. However, POS and POSITION should **not** be used in conjunction with bookmarks as bookmarks can cause these keywords, kept mainly for Series 3c compatibility, to become inaccurate. Note that if bookmarks are used in conjunction with POS and POSITION accuracy can be restored by using FIRST or LAST on the current view.

The new commands provided for the use of bookmarks are as follows: BOOKMARK puts a bookmark at the current record of the current database view. The value returned can be passed to GOTOMARK to make the record current again and to KILLMARK to delete the bookmark.

SAVING RECORDS

When using the Series 5 extensions to CREATE and OPEN, you should also use the new MODIFY, INSERT, PUT and CANCEL keywords in preference to the APPEND and UPDATE Series 3c commands. APPEND and UPDATE will still work as expected, but do not naturally fit in the DBMS model.

- MODIFY allows records to be changed without being moved to the end of the set (as UPDATE still does).
- Instead of copying the current record to the end of the set as APPEND does, INSERT appends a new record to the end of the set with numeric fields set to 0 and string fields empty if values have not been assigned to them.
- PUT marks the end of a database’s INSERT or MODIFY phase and makes the changes permanent.
- CANCEL marks the end of a database’s INSERT or MODIFY phase and discards the changes made during that phase.

THE NUMBER OF RECORDS

The COUNT function returns the number of records in the file. If you try to count the number of records between assignment and APPEND/UPDATE or between MODIFY/INSERT and PUT an ‘Incompatible update mode’ error will be raised.

CLOSING VIEWS AND DATABASES

CLOSE closes the current view on a database. If there are no other views open on the database then the database itself will be closed.

INDEXES

Indexes can be constructed on a table using several fields as *keys*. These indexes are subsequently used to provide major speed improvements when opening a table or views on them.

Further database functionality is provided in the Database OPX, discussed in the ‘‘OPX.pdf’ document.

COMPACTION

COMPACT replaces the COMPRESS command on the Series 5. This compacts a database, rewriting the file in place without any removed or deleted data. All views on the database and the hence the file itself should be closed before calling this command. Compaction may also be done automatically on closing a file by setting the automatic compaction flag using SETFLAGS. See the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document for full details of this.

OPENING A DATABASE CREATED BY THE DATA APPLICATION

It is currently not possible to open an OPL database from the Data application. You can however open a file created by the Data application in an OPL program. The file is opened for reading only, because if it were written to, OPL would have to discard all the formatting characters and prevent the Data application from reopening the file subsequently. An OPL program can create a new OPL database and copy the Data application records into it if necessary.

To open a Data application database that has one string field which you need to access, you could use:

```
OPEN “file”,a,a$
```

Types not supported by OPL will be ignored. Note that integer fields in the Data application correspond to long integer fields in OPL: the Data application does not support (16-bit) integer fields. The types and order of the OPL field handles must match the fields in the Data file. For example, if the data file `Data2` contains:

1. long integer field

2. date/time field (ignored by OPL)

3. string field

4. floating-point number field

you could access the fields supported by OPL using:

```
OPEN “Data2”,A, f1&,f2$,f3
```

It would be better, however, to use the SQL SELECT clause to name the required Data file fields explicitly. For this to be possible it is necessary to use table name and the same field names as are used by Data. All Data files have a single table called `Table1`. The fields (referred to internally as columns in Data) are named `ColA1`, `ColA2`, etc.

So, with the field types from the previous example, the Data file could be opened using:

```
OPEN “Data2 SELECT ColA1,ColA3,ColA4 FROM Table1”,a,f1&,f2$,f3
```

INDEX

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
```

➎ 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.

➎ 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

③ 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.

③ 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.

❸ 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.

❸ 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$`.

❸ 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

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

OPL

OPL & DISKS

CONTENTS

USING DISKS IN OPL

❺ On the Series 5, *memory disks* in the `D:` drive may be used in the same way as the internal memory by specifying `D:` where you would usually specify `C:`.

❸ On the Series 3c, Solid State Disks (*SSDs*), which are explained in detail in the User guide, may be used.

Siena On the Siena, there is no support for using disks.

There are two main reasons for using disks:

- To provide more room for storing data.
- To make backup copies of important information, in case you accidentally change or delete it (or even lose your Psion).

The discussion below explains the use of SSDs for OPL on the Series 3c.

❸ TYPES OF SOLID STATE DISK

There are two types - *Ram SSDs* and *Flash SSDs.* They fit into the SSD drives marked `A` and `B`, at either side of the Series 3c.

- Flash SSDs are for storing or backing up information which is infrequently changed. This includes finished OPL programs.
- Ram SSDs are for storing or backing up information which changes frequently. This includes OPL programs you are still writing or testing.

You can, though, save programs and data files to either kind of SSD, as you see fit.

❸ HOW TO PUT PROGRAMS ON AN SSD

To create a new OPL module on an SSD, use the ‘New file’ option in the System screen as before, but set the `Disk` line of the dialog to `A` or `B` as required.

To copy an OPL module on to an SSD, move onto the module name where it is listed under the Program icon, and use the ‘Copy file’ option on the ‘File’ menu. Set the ‘To file: Disk’ line to `A` or `B`. If you want this copy to have a different name to the original, type the name to use, on the ‘To file: Name’ line. The new copy will appear in the list under the Program icon, but with `[A]` or `[B]` after its name.

To copy the **translated** version of an OPL module, move onto the name in the list under the RunOpl icon (to the right of the Program icon), then proceed as before.

❸ SSDS FROM INSIDE OPL

Your OPL programs can create or use data files on SSDs. To do so, begin the name of the data file with `A:` or `B: -` for example:

```
CREATE “B:JKQ”,A,X1$,X2$
```

tries to create a data file `JKQ` on an SSD in `B`, while

```
DELETE “A:X300”
```

tries to delete a data file `X300` on an SSD in `A`.

Don’t confuse the drive names A and B with the logical names A, B, C and D. Logical names are unaffected by which drive a data file is on.

The internal memory can be referred to as M:, if required. For example:

```
PROC delx300:
  LOCAL a$(3),c%
  a$=“MAB” :c%=1        REM default to “Internal”
  dINIT “Delete X300 data file”
  dCHOICE c%,”Disk:”,”Internal,A,B”
  IF DIALOG             REM returns 0 if cancelled
    DELETE MID$(A$,c%,1)+”:X300”
  ENDIF
ENDP
```

In this example, MID$(A$,c%,1) gives “M”, “A” or “B”, according to the choice made in the dialog. This is added on the front of “:X300” to give the name of the file to delete - “M:X300”, “A:X300” or “B:X300”.

When using data files with SSDs, follow the same guidelines as with OPL programs - Flash SSDs are for one-off or “finished” information, while Ram SSDs are for information which is still being changed.

DIRECTORIES AND DOS STRUCTURE

The internal memory, memory disks and SSDs use a DOS-compatible directory structure, the same as that used by disks on PCs. For more details, see the ‘Advanced.pdf’ document.

OPL

EXAMPLE PROGRAMS

CONTENTS

OPL

This document contains example programs written in OPL. The programs are not intended to demonstrate all the features of OPL, but they should give you a few hints. To find out more about a particular command or function, refer to the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document.

There are some further example programs in the ‘Advanced Topics’ section of the ‘Advanced.pdf’ document.

WHEN YOU’RE TYPING IN

- You can type procedures in all uppercase, all lowercase or any mixture of the two. Be careful with character codes, though - %A is different to %a.

- When there is more than one command or function on a line, separate each one with a space and colon - for example:
  `CLS :PRINT “hello” :GET`
  However, the colon is optional before a `REM` statement for example:
  `CLS REM Clears the screen`
  and
  `CLS :REM Clears the screen`
  are both OK.

- Put a space between a command and the arguments which follow it - for example `PRINT a$`. But don’t put a space between a function and the arguments in brackets - for example `CHR$(16)`.

- It doesn’t matter how many spaces or tabs you have at the beginning of a line.

ERRORS

The following programs do not include full error handling code. This means that they are shorter and easier to understand, but may fail if, for example, you enter the wrong type of input to a variable.

If you want to develop other programs from these example programs, it is recommended that you add some error handling code to them. See the ‘Error Handling’ section of the ‘Advanced.pdf’ document for further details.

COUNTDOWN TIMER

❺ For the Series 5:

```
PROC timer:
   LOCAL min&,sec&,secs&,i%
   sec&=1
   dINIT “Countdown timer”
   dLONG min&,“Minutes”,0,59
   dLONG sec&,“Seconds”,0,59
   dBUTTONS “Cancel”,-27,“Start”,%s
   IF DIALOG=%s
     FONT 12,16
     secs&=sec&+60*min&
     WHILE secs&
          PAUSE -20                   REM a key gets us out
          IF KEY
               RETURN
          ENDIF
          secs&=secs&-1
          AT 20,6 :PRINT NUM$(secs&/60,-2);“m”
          AT 24,6 :PRINT NUM$(mod&:(secs&,int(60)),-2);“s”
     ENDWH
     DO
          BEEP 5,300
          PAUSE 10
          IF KEY :BREAK :ENDIF
          i%=i%+1
     UNTIL i%=10
  ENDIF
ENDP

PROC mod&:(a&,b&)
  REM modulo function
  REM computes (a&)mod(b&)
  RETURN a&-(a&/b&)*b&
ENDP
```

❸ For the Series 3c and Siena:

```
PROC timer:
  LOCAL min&,sec&,secs&,i%
  CACHE 2000,2000
  sec&=1
  dINIT “Countdown timer”
  dLONG min&,”Minutes”,0,59
  dLONG sec&,”Seconds”,0,59
  dBUTTONS “Cancel”,-27,”Start”,13
  IF DIALOG=13
     STATUSWIN ON
     FONT 11,16
     secs&=sec&+60*min&
     WHILE secs&
          PAUSE -20                   REM a key gets us out
          IF KEY
               RETURN
          ENDIF
          secs&=secs&-1
          AT 20,6 :PRINT NUM$(secs&/60,-2);”m”
          AT 24,6 :PRINT NUM$(mod&:(secs&,int(60)),-2);”s”
     ENDWH
     DO
          BEEP 5,300
          PAUSE 10
          IF KEY :BREAK :ENDIF
          i%=i%+1
     UNTIL i%=10
  ENDIF
ENDP
```

```
PROC mod&:(a&,b&)
  REM modulo function
  REM computes (a&)mod(b&)
  RETURN a&-(a&/b&)*b&
ENDP
```

DICE

When the program is run, a message is displayed saying that the dice is rolling. You then press a key to stop it. A random number from one to six is displayed and you choose whether to roll again or not.

```
PROC dice:
  LOCAL dice%
  DO
     CLS :PRINT “DICE ROLLING:”
     AT 1,3 :PRINT “Press a key to stop”
     DO
          dice%=(RND*6+1)
          AT 1,2 :PRINT dice%
     UNTIL KEY
     BEEP 5,300
     dINIT “Roll again?”
     dBUTTONS “No”,%N,“Yes”,%Y
  UNTIL DIALOG<>%y
ENDP
```

RANDOM NUMBERS

In this example, the RND function returns a random floating-point number, between 0 and 0.9999999... It is then multiplied by 6, and 1 is added, to give a number from 1 to 6.9999999... This is rounded down to a whole number (from 1 to 6) by assigning to the integer `dice%`.

BIRTHDAYS

This procedure finds out on which day of the week people were born.

```
PROC Birthday:
  LOCAL day&,month&,year&,DayInWk%
  DO
     dINIT
     dTEXT “”,“Enter your date of birth”,2
     dTEXT “”,“Use numbers, eg 23 12 1963”,$202
     dLONG day&,“Day”,1,31
     dLONG month&,“Month”,1,12
     dLONG year&,“Year”,1900,2155
     IF DIALOG=0
          BREAK
     ENDIF
     DayInWk%=DOW(day&,month&,year&)
     CLS :PRINT DAYNAME$(DayInWk%),day&,month&,year&
```

```
    dINIT “Again?”
    dBUTTONS “No”,%N,“Yes”,%Y
  UNTIL DIALOG<>%y
ENDP
```

The DOW function works out what day of the week, from 1 to 7, a date is. The DAYNAME$ function then converts this to MON, TUE and so on. MON is 1 and SUN is 7.

DATA FILES

The following module works on a data file called `DATA`, containing names, addresses, post codes and telephone numbers. It assumes this file has already been created with a statement like this:

```
CREATE “DATA”,A,nm$,ad1$,ad2$,ad3$,ad4$,tel$
```

❺ To use a database created with the Data application, see the ‘Series 5 Database Handling’ section of the ‘Database.pdf’ document.

❸ To use the `DATA` file which the Database application uses, you need to open `“\DAT\DATA.DBF”`.

The first procedure is the controlling, calling procedure, offering you choices. The next two let you add or edit records.

```
PROC files:
  GLOBAL nm$(255),ad1$(255),ad2$(255)
  GLOBAL ad3$(255),ad4$(255),tel$(255),title$(30)
  LOCAL g%
  OPEN “DATA”,A,nm$,ad1$,ad2$,ad3$,ad4$,tel$
  DO
    CLS
    dINIT “Select action”
    REM !!Swap prompt and body in dTEXT for Series 3c and Siena!!
    dTEXT “Add new record”,“”,$402
    dTEXT “Find and edit a record”,“”,$402
    g%=DIALOG
    IF g%=2
         add:
    ELSEIF g%=3
         edit:
    ENDIF
  UNTIL g%=0
  CLOSE
ENDP

PROC add:
  nm$=“” :ad1$=“” :ad2$=“”
  ad3$=“” :ad4$=“” :tel$=“”
  title$=“Enter a new record”
  IF showd%:
    APPEND
  ENDIF
ENDP
```

```
PROC edit:
  LOCAL search$(30),p%
  dINIT “Find and edit a record”
  dEDIT search$,“Search string”,15
  IF DIALOG
     FIRST
     IF FIND(“*”+search$+“*”)=0
          ALERT(“No matching records”)
          RETURN
     ENDIF
     DO
          nm$=A.nm$ :ad1$=A.ad1$ :ad2$=A.ad2$
          ad3$=A.ad3$ :ad4$=A.ad4$ :tel$=A.tel$
          title$=“Edit matching record”
          IF showd%:
               UPDATE :BREAK
          ELSE
               NEXT
          ENDIF
          FIND(“*”+search$+“*”)
          IF EOF
               ALERT(“No more matching records”)
               BREAK
          ENDIF
     UNTIL 0
  ENDIF
ENDP

PROC showd%:
  LOCAL ret%
  dINIT title$
  dEDIT nm$,“Name”,25
  dEDIT ad1$,“Street”,25
  dEDIT ad2$,“Town”,25
  dEDIT ad3$,“County”,25
  dEDIT ad4$,“Postcode”,25
  dEDIT tel$,“Phone”,25
  ret%=DIALOG
  IF ret%
     A.nm$=nm$ :A.ad1$=ad1$ :A.ad2$=ad2$
     A.ad3$=ad3$ :A.ad4$=ad4$ :A.tel$=tel$
  ENDIF
  RETURN ret%
ENDP
```

RE-ORDER

When you use the Data application and enter or change an entry, it goes to the end of the database file. However, **if**, in your address book, each entry begins with a person's second name - for example, `Tate, Hazel` - you can use this program to re-order all of the entries. This doesn't change the way you find an entry, but after running it you can step through it like a paper address book, or print it out neatly ordered.

This procedure can be used as required for any data file in internal memory or on memory disk for the Series 5 or on Ram SSDs for the Series 3c. For the Series 3c, note that if used on a data file held on a Flash SSD it would use up disk space each time you run it. The dialog it shows is set to show data files used by Data.

You can adapt this procedure to sort other types of data files in other ways.

Note that on the Series 5, this would be better done with the more advanced features available in the Database OPX. See the 'Using OPXs on the Series 5' section of the 'Advanced.pdf' document for more details of this. You could also use restriction of files by UID in the dFILE keyword to restrict to databases only.

```
PROC reorder:
  LOCAL last%,e$(255),e%,lpos%,n$(255),c%
  n$="\dat\*.dbf"
  dINIT "Re-order Data file"
  dFILE n$,"Filename",0
  IF DIALOG                      REM returns 0 if cancelled
    OPEN n$,a,a$
    LAST :last%=POS
    IF COUNT>0
        WHILE last%<>0
            POSITION last% :e%=POS
            e$=UPPER$(a.a$)
            DO
                IF UPPER$(a.a$)<e$
                    e$=UPPER$(a.a$) :e%=POS
                ENDIF
                lpos%=POS :BACK
            UNTIL pos=1 and lpos%=1
            POSITION e%
            PRINT e$
            UPDATE :last%=last%-1
        ENDWH
    ENDIF
    CLOSE
  ENDIF
  GET
ENDP
```

If you try to reorder a file which is already open (i.e. shown in bold on the System screen) you will see a "*File*' is in use' ('File or device in use' on the Series 3c) error. You should close the file and then try again.

STOPWATCH

Here is a simple stopwatch with lap times. Note that the Psion switches off automatically after a time if no keys are pressed; you may want to disable this feature (from the Control Panel in the System screen on the Series 5 or with the ‘Auto switch off’ option in the System screen on the Series 3c) before running this program.

Each timing is only accurate to within one second, as the procedure is based on the SECOND function.

```
PROC watch:
  LOCAL k%,s%,se%,mi%
  FONT 11,16
  AT 20,1 :PRINT “Stopwatch”
  AT 15,11 :PRINT “Press a key to start”
  GET
  DO
    CLS :mi%=0 :se%=0 :s%=SECOND
    AT 15,11 :PRINT “  S=Stop, L=Lap  ”
loop::
  k%=KEY AND $ffdf              REM ensures upper case
  IF k%=%S
    GOTO pause::
  ENDIF
  IF k%=%L
    AT 20,6 :PRINT “Lap: ”;mi%;“:”;
    IF se%<10 :PRINT “0”; :ENDIF
    PRINT se%;“ ”;
  ENDIF
  IF SECOND<>s%
    s%=SECOND :se%=se%+1
    IF se%=60 :se%=0 :mi%=mi%+1 :ENDIF
    AT 17,8
    PRINT “Mins”,mi%,“Secs”,
    IF se%<10 :PRINT “0”; :ENDIF
    PRINT se%;“ ”;
  ENDIF
  GOTO loop::
pause::
  mINIT
  mCARD “Watch”,“Restart”,%r,“Zero”,%z,“Exit”,%x
  k%=MENU
  IF k%=%r
    GOTO loop::
  ENDIF
  UNTIL k%<>%z
ENDP
```

INSERTING A NEW LINE IN A DATABASE

If you insert a new label in a database, the entries will no longer match up with the labels. Rather than using the ‘Update’ option on every entry, to insert a suitable blank line in each one, you can use this program to do this for the entire data file.

The Data application allows you to use as many lines (fields) as you want in an entry (record); OPL can only access 32 fields. This program only lets you insert a new field in the first 16 fields, although you can adapt the program simply to check up to 31 fields.

If, in Data, you enter a line longer than 255 characters, it is stored as two fields, with a character of code 20 at the start of the second field. This program correctly handles any such fields.

The program checks that the 17th field is blank, as it will be overwritten by what was the 16th field. If a long entry has a 17th field, and it contains text, the program skips this entry. The rest of longer entries - even if there are more than 32 fields will be unchanged.

If you insert a new field at a position below the last label, Data will not show it, even when using ‘Update’.

The maximum record length in OPL is 1022 characters. The OPEN command will display a ‘Record too large’ error if the file contains a record longer than this.

```
PROC label:
   LOCAL a%,b%,c%,d%,s$(128),s&,i$(17,255)
   s$=“\dat\*.dbf”
   dINIT “Insert new field”
   dFILE s$,“Data file”,0
   dLONG s&,“Break at line (1-16)”,1,16
   IF DIALOG
     OPEN s$,A,a$,b$,c$,d$,e$,f$,g$,h$,i$,j$,k$,l$,m$,n$,o$,p$,q$
     c%=COUNT :a%=1
     WHILE a%<=c%
          AT 1,1 :PRINT “Entry”,a%,”of”,c%,
          IF A.q$=“”                   REM Entry (hopefully) not too long
              i$(1)=A.a$ :i$(2)=A.b$ :i$(3)=A.c$ :i$(4)=A.d$
              i$(5)=A.e$ :i$(6)=A.f$ :i$(7)=A.g$ :i$(8)=A.h$
              i$(9)=A.i$ :i$(10)=A.j$ :i$(11)=A.k$ :i$(12)=A.l$
              i$(13)=A.m$ :i$(14)=A.n$ :i$(15)=A.o$ :i$(16)=A.p$
              d%=0 :b%=0
              WHILE d%<s&+b%          REM find field to break at
                   d%=d%+1
                   IF LEFT$(i$(d%),1)=CHR$(20)     REM line>255...
                        b%=b%+1       REM ...so it’s 2 fields
                   ENDIF
              ENDWH
              b%=17
              WHILE b%>d%             REM copy the fields down
                   i$(b%)=i$(b%-1) :b%=b%-1
              ENDWH
              i$(d%)=“”               REM and make an empty field
              A.a$=i$(1) :A.b$=i$(2) :A.c$=i$(3) :A.d$=i$(4)
              A.e$=i$(5) :A.f$=i$(6) :A.g$=i$(7) :A.h$=i$(8)
              A.i$=i$(9) :A.j$=i$(10) :A.k$=i$(11) :A.l$=i$(12)
              A.m$=i$(13) :A.n$=i$(14) :A.o$=i$(15) :A.p$=i$(16)
              A.q$=i$(17)
          ELSE
              PRINT “has too many fields”
              PRINT “Press a key...” :GET
          ENDIF
          UPDATE :FIRST
          a%=a%+1
     ENDWH :CLOSE
  ENDIF
ENDP
```

BOUNCING BALL

```
PROC bounce:
  LOCAL posX%,posY%,changeX%,changeY%,k%
  LOCAL scrx%,scry%,info%(10)
  SCREENINFO info%()
  scrx%=info%(3) :scry%=info%(4)
  posX%=1 :posY%=1
  changeX%=1 :changeY%=1
  DO
     posX%=posX%+changeX%
     posY%=posY%+changeY%
     IF posX%=1 OR posX%=scrx%
          changeX%=-changeX%
          REM at edge ball changes direction
          BEEP 2,600                    REM low beep
     ENDIF
     IF posY%=1 or posY%=scry%        REM same for y
          changeY%=-changeY%
          BEEP 2,200                    REM high beep
     ENDIF
     AT posX%,posY% :PRINT “0”;
     PAUSE 2                          REM Try changing this
     AT posX%,posY% :PRINT “ ”;       REM removes old ‘0’ character
     k%=KEY
  UNTIL k%
ENDP
```

CIRCLES

❺ Here is an example program for drawing circles or ellipses, filled or unfilled for the Series 5:

```
PROC draw:
  LOCAL d%
  DO
     dINIT “Draw a circle or an ellipse?”
     dBUTTONS “Circle”,%c OR $200,“Ellipse”,%e OR $200,“Cancel”,-27
     d%=DIALOG
     IF d%=%c
          circle:
     ELSEIF d%=%e
```

```
         ellipse:
     ENDIF
   UNTIL d%=0
ENDP

PROC circle:
   LOCAL x&,y&,r&,f%
   dINIT “Drawing parameters”
   x&=320 :dLONG x&,”Centre x position”,0,639
   y&=120 :dLONG y&,”Centre y position”,0,249
   r&=20 :dLONG r&,”Radius”,1,320
   f%=0 :dCHECKBOX f%,”Filled”
   dBUTTONS “Draw”,%d,“Cancel”,-27
   IF DIALOG
     gAT x&,y&
     gCIRCLE r&,f%
     GET
     gCLS
   ENDIF
ENDP

PROC ellipse:
   LOCAL x&,y&,hr&,vr&,f%
   dINIT “Drawing parameters”
   x&=320 :dLONG x&,”Centre x position”,0,639
   y&=120 :dLONG y&,”Centre y position”,0,249
   hr&=20 :dLONG hr&,”Horizontal Radius”,1,320
   vr&=20 :dLONG vr&,”Vertical Radius”,1,320
   f%=0 :dCHECKBOX f%,”Filled”
   dBUTTONS “Draw”,%d,“Cancel”,-27
   IF DIALOG
     gAT x&,y&
     gELLIPSE hr&,vr&,f%
     GET
     gCLS
   ENDIF
ENDP
```

❸ Here are **two** example programs for drawing circles - the first hollow, the second filled for the Series 3c and Siena:

```
PROC circle:
   LOCAL a%(963),c&,d%,x&,y&,r&,h,y%,y1%,c2%
   dINIT “Draw a circle”
   x&=240 :dLONG x&,“Centre x pos”,0,479
   y&=80 :dLONG y&,“Centre y pos”,0,159
   r&=20 :dLONG r&,“Radius”,1,120
   h=1 :dFLOAT h,“Relative height”,0,999
   IF DIALOG
     a%(1)=x&+r& :a%(2)=y& :a%(3)=4*r&
     c&=1 :d%=2*r& :y1%=0
```

```
    WHILE c&<=d%
         c2%=c&*2 :y%=-SQR(r&*c2%-c&**2)*h
         a%(2+c2%)=-2 :a%(3+c2%)=y%-y1%
         y1%=y% :c&=c&+1
    ENDWH
    c&=1
    WHILE c&<=d%
         c2%=c&*2 :y%=SQR(r&*c2%-c&**2)*h
         a%(2+a%(3)+c2%)=2 :a%(3+a%(3)+c2%)=y%-y1%
         y1%=y% :c&=c&+1
    ENDWH
    gPOLY a%()
  ENDIF
ENDP

PROC circlef:
  LOCAL c&,d%,x&,y&,r&,h,y%
  dINIT “Draw a filled circle”
  x&=240 :dLONG x&,“Centre x pos”,0,479
  y&=80 :dLONG y&,“Centre y pos”,0,159
  r&=20 :dLONG r&,“Radius”,1,120
  h=1 :dFLOAT h,“Relative height”,0,999
  IF DIALOG
    c&=1 :d%=2*r& :gAT x&-r&,y& :gLINEBY 0,0
    WHILE c&<=d%
         y%=-SQR(r&*c&*2-c&**2)*h
         gAT x&-r&+c&,y&-y% :gLINEBY 0,2*y%
         c&=c&+1
    ENDWH
  ENDIF
ENDP
```

If you use gUPDATE OFF after the IF DIALOG line, and gUPDATE ON before the ENDIF, the procedure will run a little faster. However, all but the smaller circles will be drawn rather jerkily, piece by piece.

❸ ZOOMING

This is an example for the Series 3c only. The Series 5 does not have status windows and the Siena does not have large status windows owing to its screen size.

For each of the three types of status window, this program changes the font to implement zooming.

Press Psion-Z to cycle between small, medium and large fonts, and Shift-Psion-Z to cycle in the other direction. Esc changes to the next status window.

As well as changing the font and style for the text window (for PRINT etc.), the FONT command automatically changes the default graphics window size (ID=1) and the text window size to fit exactly in the space left by any status window. (A special feature not used here is that `FONT -$3fff,0` just changes the window sizes **without** changing font).

The procedure `dispinfo:` uses the command SCREENINFO to display the margin sizes in pixels between the default window and the text window, the text screen size in character units, and the text screen's character width and line height in pixels.

```
PROC tzoom:
  STATUSWIN OFF                 REM no status window
  zoom:                         REM display with zooming
  STATUSWIN ON,2                REM large status window
  zoom:
  STATUSWIN ON,1                REM and small
  zoom:
ENDP

PROC zoom:
  LOCAL font%(3),font$(3,20),style%(3)
  LOCAL g%,km%,zoom%
  zoom%=1
  font%(1)=13 :font$(1)=“(Mono 6x6)” :style%(1)=0
  font%(2)=4 :font$(2)=“(Mono 8x8)” :style%(2)=0
  font%(3)=12 :font$(3)=“(Swiss 16)” :style%(3)=16
  g%=%z+$200
  DO
    IF g%=%z+$200
        IF km% AND 2            REM Shift-PSION-Z
            zoom%=zoom%-1
            IF zoom%<1 :zoom%=3 :ENDIF
        ELSE                    REM PSION-Z
            zoom%=zoom%+1
            IF zoom%>3 :zoom%=1 :ENDIF
        ENDIF
        FONT font%(zoom%),style%(zoom%)
        PRINT “Font=”;font%(zoom%),font$(zoom%),
        PRINT “Style=”;style%(zoom%)
        dispinfo:
        PRINT rept$(“1234567890”,15)
        gBORDER 0
    ENDIF
    g%=GET
    km%=KMOD
  UNTIL g%=27
ENDP

PROC dispinfo:
  LOCAL scrInfo%(10)
  SCREENINFO scrInfo%()
  PRINT “Left margin=”;scrInfo%(1),
  AT 17,2 :PRINT “Top margin=”;scrInfo%(2)
  PRINT “Screen width=”;scrInfo%(3)
  AT 17,3 :PRINT “Screen height=”;scrInfo%(4)
  PRINT “Char width=”;scrInfo%(7)
  AT 17,4 :PRINT “Line height=”;scrInfo%(8)
ENDP
```

ANIMATION EXAMPLE

This program requires five bitmap files - `one.pic` to `five.pic`. Each of these would differ slightly. They might, for example, be five ‘snapshots’ of a running human figure, each with the legs at a different point in their cycle.

The program copies each bitmap into a window of its own, then makes each window visible in turn, each time slightly further across the screen.

To make bitmap files, first draw the pattern you want with any of the graphics drawing commands. (Use `gLINEBY 0,0` to draw single dots.) When the pattern is complete, use gSAVEBIT to make the bitmap file. For advanced animation, you could use a sprite as described in the ‘Using OPXs on the Series 5’ section of the ‘Advanced.pdf’ document for the Series 5, and as described in the ‘Advanced Topics’ section of the ‘Advanced.pdf’ document for the Series 3c and Siena.

```
PROC animate:
  LOCAL id%(5),i%,j%,s$(5,10),w%,h%,edge%
  REM example width and height
  w%=16 :h%=28
  REM screen edge - use 480 for Series 3c and 240 for Siena
  edge%=640
  REM need not have “.pic” in the following for Series 3c and Siena
  s$(1)=“one.pic” :s$(2)=“two.pic” :s$(3)=“three.pic”
  s$(4)=“four.pic” :s$(5)=“five.pic” :j%=1
  WHILE j%<6
    i%=gLOADBIT(s$(j%))
    id%(j%)=gCREATE(0,0,w%,h%,0)
    gCOPY i%,0,0,w%,h%,3
    gCLOSE i% :j%=j%+1
  ENDWH
  i%=0 :gORDER 1,9
  DO
    j%=(i%-5*(i%/5))+1          REM (i% MOD 5)+1
    gVISIBLE OFF                REM previous window
    gUSE id%(j%)                REM new window
    gSETWIN i%,20               REM position it
    gORDER id%(j%),1            REM make foreground
    gVISIBLE ON                 REM make visible
    i%=i%+1 :PAUSE 2
  UNTIL KEY OR (i%>(edge%-w%))
ENDP
```

❸ TWO-VOICE “ICE-CREAM VAN” SOUND

This example is for the Series 3c and Siena only.

The following program plays a rising and falling scale. It uses the amplifier-driven loudspeaker device (with device driver SND:) which allows you to play tunes using two-note chords - ie it has two *voices*.

This program uses I/O keywords as described in the ‘Advanced Topics’ section. Take care to enter them exactly as shown here.

```
PROC main:
   LOCAL ret%,sndHand%
   ret%=IOOPEN(sndHand%,“SND:”,-1)     REM open the device
   IF ret%<0
      PRINT “Failed to start”
      PRINT err$(err)
      GET
   ELSE
      icecream:(sndHand%)
      IOCLOSE(sndHand%)
   ENDIF
ENDP

PROC icecream:(sndHand%)
   LOCAL notes1%(4),notes2%(14)
   LOCAL s1stat%,len1%,len2%
   REM define 1st voice
   notes1%(1)=1048 :notes1%(2)=96      REM freq, duration
   notes1%(3)=524  :notes1%(4)=48
   len1%=2                             REM number of notes in voice 1
   REM define 2nd voice
   notes2%(1)=1048 :notes2%(2)=16
   notes2%(3)=1320 :notes2%(4)=16
   notes2%(5)=1568 :notes2%(6)=16
   notes2%(7)=2092 :notes2%(8)=16
   notes2%(9)=1568 :notes2%(10)=16
   notes2%(11)=1320 :notes2%(12)=16
   notes2%(13)=1048 :notes2%(14)=48
   len2%=7                             REM number of notes in voice 2
   IOC(sndhand%,1,s1stat%,notes1%(),len1%)
   REM voice 1 asynchronous
   IOW(sndHand%,2,notes2%(),len2%)
   REM voice 2 synchronous
   IOWAITSTAT s1stat%
ENDP
```

`notes1%()` and `notes2%()` are set up to hold `len1%` and `len2%` notes to be played on voice 1 and voice 2 respectively. The number of notes to each voice must not exceed 16384.

Each note is composed of two consecutive integers in the array with the first of each pair giving the frequency in Hz (middle A is 440Hz) and the second giving the note duration in quarter-beats per minute.

INDEX

OPL

ERROR HANDLING

CONTENTS

SYNTAX ERRORS

Syntax errors are those which are reported when translating a procedure. (Other errors can occur while you’re running a program.) The OPL translator will return you to the line where the first syntax error is detected.

All programming languages are very particular about the way commands and functions are used, especially in the way program statements are laid out. Below are a number of errors which are easy to make in OPL. The incorrect statements are in bold and the correct versions are on the right.

PUNCTUATION ERRORS

Omitting the colon between statements on a multi-statement line:

Incorrect	*Correct*
a$=“text” **PRINT a$**	a$=“text” :PRINT a$

Omitting the space before the colon between statements:

Incorrect	*Correct*
a$=b$**:PRINT a$**	a$=b$:PRINT a$

Omitting the colon after a called procedure name:

Incorrect	*Correct*
PROC proc1:	PROC proc1:
GLOBAL a,b,c	GLOBAL a,b,c
.	.
proc2	proc2:
ENDP	ENDP

Using only 1 colon after a label in GOTO/ONERR/VECTOR (instead of 0 or 2):

Incorrect	*Correct*
GOTO **below:**	GOTO below
.	.
below::	below::

STRUCTURE ERRORS

The DO...UNTIL, WHILE...ENDWH and IF...ENDIF structures can produce a ‘Structure fault’ error if used incorrectly:

- Mixing up the three structures - e.g. by using DO...WHILE instead of DO...UNTIL.
- Using BREAK or CONTINUE in the wrong place.
- Using ELSE IF with a space, instead of ELSEIF.
- VECTOR...ENDV can also produce a ‘Structure fault’ error if used incorrectly.

Attempting to nest any combination of these structures more than eight levels deep will produce a ‘Too complex’ error.

ERRORS IN RUNNING PROCEDURES

OPL may display an error message and stop a running program if certain ‘error’ conditions occur. This may happen because:

- There is a mistake, or *bug*, in your program, which could not be detected during translation - for example, a calculation has involved a division by zero.
- A problem has occurred which prevents a command or function from working - for example, an APPEND command may fail because a disk is full.

Unless you include statements which can handle such errors when they occur, OPL will use its own error handling mechanism. The program will stop and an error message be displayed. The first line gives the names of the procedure in which the error occurred, and the module this procedure is in. The second line is the ‘error message’ - one of the messages listed at the end of this section. If appropriate, you will also see a list of variable names or procedure names causing the error.

If you were editing the module with the Program editor and you ran it from there, you would also be taken back to editing the OPL module, with the cursor at the line where the error occurred.

ERROR HANDLING FUNCTIONS AND COMMANDS

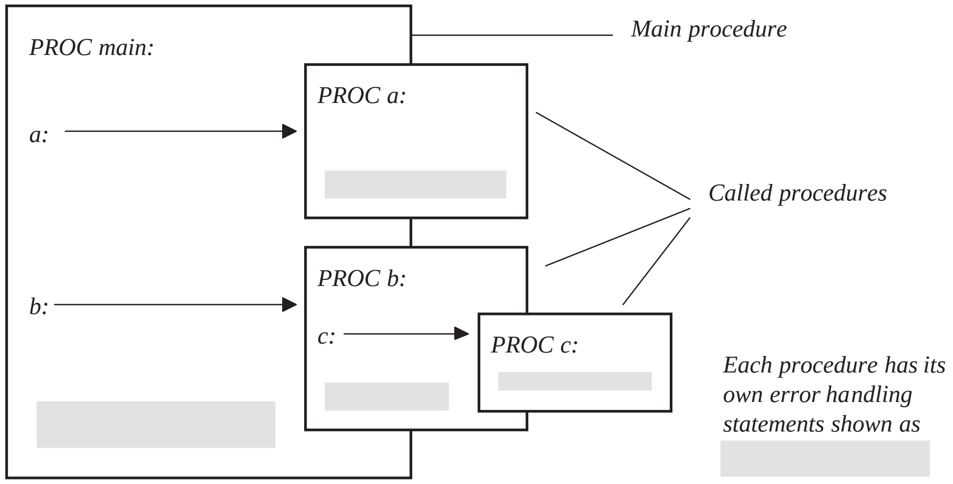
To prevent your program being stopped by OPL when an error occurs, include statements in your program which anticipate possible errors and take appropriate action. The following error handling facilities are available in OPL:

- TRAP temporarily suppresses OPL’s error processing.
- ERR and ERR$ (and ERRX$ on the Series 5) find out what kind of error has occurred.
- ONERR establishes an error handler which can suppress OPL’s error processing over whole modules.
- RAISE can be used to simulate error conditions.

These facilities put you in control and must be used carefully.

STRATEGY

You should design the error handling of a program in the same way as the program itself. OPL works best when programs are built up from procedures, and you should design your error handling on the same basis. Each procedure should normally contain its own local error handling:



The error handling statements can then be appropriate to the procedure. For example, a procedure which performs a calculation would have one type of error handling, but another procedure which offers a set of choices would have another.

TRAP

❺ TRAP can be used with any of these commands: APPEND, BACK, CANCEL, CLOSE, COPY, CREATE, DELETE, ERASE, EDIT, FIRST, gCLOSE, gCOPY, gFONT, gPATT, gSAVEBIT, gUNLOADFONT, gUSE, INPUT, INSERT, LAST, LCLOSE, LOADM, LOPEN, MKDIR, MODIFY, NEXT, OPEN, OPENR, POSITION, PUT, RAISE (see below), RENAME, RMDIR, UNLOADM, UPDATE and USE.

❸ TRAP can be used with any of these commands: APPEND, BACK, CACHE, CLOSE, COMPRESS, COPY, CREATE, DELETE, ERASE, EDIT, FIRST, gCLOSE, gCOPY, gFONT, gPATT, gSAVEBIT, gUNLOADFONT, gUSE, INPUT, LAST, LCLOSE, LOADM, LOPEN, MKDIR, NEXT, OPEN, OPENR, POSITION, RENAME, RMDIR, UNLOADM, UPDATE and USE.

The TRAP command immediately precedes any of these commands, separated from it by a space - for example:

```
TRAP INPUT a%
```

If an error occurs in the execution of the command, the program does not stop, and the next line of the program executes as if there had been no error. Normally you would use ERR on the line after the TRAP to find out what the error was.

EXAMPLE

When INPUT is used without TRAP and a text string is entered when a number is required, the display just scrolls up and a ? is shown, prompting for another entry. With TRAP in front of INPUT, you can handle bad entries yourself:

```
PROC trapinp:
  LOCAL profit%
  DO
  PRINT
  PRINT “Enter profit”,
  TRAP INPUT profit%
  UNTIL ERR=0
  PRINT “Valid number”
  GET
ENDP
```

This example uses the ERR function, described next.

ERR, ERR$ AND ERRX$

When an error occurs in a program, check what number the error was, with the ERR function:

```
e%=ERR
```

If ERR returns zero, there was no error. The value returned by ERR is the number of the last error which occurred it changes when a new error occurs. TRAP sets ERR to zero if no error occurred. Check the number it returns against the error messages listed at the end of this section.

The ERR$ function gives you the message for error number `e%`:

```
e$=ERR$(e%)
```

You can also use ERR and ERR$ together:

```
e$=ERR$(ERR)
```

This returns the error message for the most recent error.

❺ The ERRX$ function gives you the extended message for the current error:

```
e$=ERRX$
```

For example, ‘Error in *MODULE\PROCEDURE*,*EXTERN1*,*EXTERN2*,...’. This is the message which would have been presented as an alert if the error had not been trapped. The use of this function gives the list of missing externals and procedure names when an error has been trapped.

EXAMPLE

The lines below anticipate that error number -101 (‘File already open’) may occur. If it does, an appropriate message is displayed.

```
TRAP OPEN “main”,A,a$
e%=ERR
IF e%              REM Checks for an error
  IF e%=-101
    PRINT “File is already open!”
  ELSE
```

```
    PRINT ERR$(e%)
  ENDIF
ENDIF
```

The inner IF...ENDIF structure displays either the message in quotes if the error was number -101, or the standard error message for any other error.

TRAP INPUT/EDIT AND THE ESC KEY

If in response to a `TRAP INPUT` or `TRAP EDIT` statement, the Esc key is pressed while no text is on the input/edit line, the 'Escape key pressed' error (number -114) will be raised. (This error will only be raised if the INPUT or EDIT has been trapped. Otherwise, the Esc key still leaves you editing.)

You can use this feature to enable someone to press the Esc key to escape from editing or inputting a value. For example:

```
PROC trapInp:
  LOCAL a%,b%,c%
  PRINT "Enter values."
  PRINT "Press Esc to exit"
  PRINT "a% =", :TRAP INPUT a% :PRINT
  IF ERR=-114 :GOTO end :ENDIF
  PRINT "b% =", :TRAP INPUT b% :PRINT
  IF ERR=-114 :GOTO end :ENDIF
    PRINT "a%*b% =",a%*b%
    PAUSE -40
    RETURN
end::
  PRINT :PRINT "OK, finishing..."
  PAUSE -40
  RETURN
ENDP
```

ONERR...ONERR OFF

ONERR sets up an error handler. This means that, whenever an error occurs in the procedure containing ONERR, the program will jump to a specified label instead of stopping in the normal way. This error handler is active until an ONERR OFF statement.

You specify the label after the word ONERR.

The label itself can then occur anywhere in the same procedure - even above the ONERR statement. After the label should come the statements handling whatever error may have caused the program to jump there. For example, you could just have the statement `PRINT ERR$(ERR)` to display the message for whatever error occurred.

All statements after the ONERR command, including those in procedures called by the procedure containing the ONERR, are protected by the ONERR, until the ONERR OFF instruction is given.

EXAMPLE

```
PROC div0:
  ONERR errHand
  PRINT 1/0
  REM cause divide by zero error -8
  RETURN    REM don’t get to this line
errHand::
  ONERR OFF
  PRINT “Error:”;err,err$(err)
  IF ERR=-8
    REM divide by zero error = -8
    PRINT “Division by zero is illegal”
  ENDIF
  GET
ENDP
```

Statements protected by ONERR

If an error occurs in the lines between `ONERR errHand` and `ONERR OFF`, the program jumps to the label `errHand::` where a message is displayed.

Always cancel ONERR with ONERR OFF immediately after the label.

WHEN TO USE ONERR OFF

You could protect the whole of your program with a single ONERR. However, it’s often easier to manage a set of procedures which each have their own ONERR...ONERR OFF handlers, each covering their own procedure. Secondly, an endless loop may occur if all errors feed back to the same single label.

For example, the diagram below shows how an error handler is left active by mistake. Two completely different errors cause a jump to the same label, and cause an inappropriate explanatory message to be displayed. In this example an endless loop is created because `next:` is called repeatedly:

```
PROC first:
  ONERR label
  a=log(-1)
  ...
  label::
  PRINT “Log error”
  next:
ENDP

PROC next:
  PRINT 2/0
  ONERR OFF
ENDP
```

MULTIPLE ONERRS

You can have more than one ONERR in a procedure, but only the most recent ONERR is active. Any errors cause a jump to the label for the most recent ONERR.

ONERR OFF disables *all* ONERRs **in the current procedure**. If there are ONERRs in **other** procedures above this procedure (*calling procedures*) these ONERRs are not disabled.

TRAP AND ONERR

TRAP has priority over ONERR. In other words, an error from a command used with TRAP will not cause a jump to the error handler specified with ONERR.

RAISE

The RAISE command generates an error, in the same way that OPL raises errors whenever it meets certain conditions which it recognises as unacceptable (for example, when invalid arguments are passed to a function). Once an error has been raised, either by OPL itself or by the RAISE command, the error-handling mechanism currently in use takes effect - the program will stop and report a message, or if you've used ONERR the program will jump to the ONERR label.

There are two reasons for using RAISE:

- You may want to mimic OPL's error conditions in your own procedures. For example, if you create a new procedure which performs a calculation and returns a value, you may want to RAISE an ‘Overflow’ or ‘Divide by zero’ error if unsuitable numbers are passed as parameters.
  In this case, you would RAISE one of the standard error numbers. You could handle this yourself with ONERR, or let OPL handle it in the normal way.

- OPL raises only a limited range of errors for general use, and you may want to raise new kinds of error codes specific to your program or particular circumstances.
  In this case, you would RAISE a new error number. With ONERR on, RAISE would go to the ONERR label, where you would have code to interpret your new error numbers. You could then display appropriate messages.
  You can use any positive number (from 0 to 127) as a new error code. Do not use any of the numbers in the list that follows.

You may also find RAISE useful for testing your error handling.

EXAMPLE

```
PROC main:
  REM calling procedure
  PRINT myfunc:(0.0)                   REM will raise error -2
ENDP


PROC myfunc:(x)
  LOCAL s
  REM returns 1/sqr(x)
  s=SQR(x)
  IF s=0
    RAISE -2
    REM 'Invalid arguments'
    REM avoids 'divide by zero'
  ENDIF
  RETURN (1/s)
ENDP
```

This uses RAISE to raise the 'Invalid arguments' error not the 'Divide by zero' error, since the former is the more appropriate message.

❺ TRAP RAISE ERR%

`TRAP RAISE err%` can be used to clear the TRAP flag and sets ERR value to `err%`. For example, using `err%=0` will clear ERR.

ERROR MESSAGES

These are the numbers of the errors which OPL can raise, and the message associated with them:

Number	*Message*
-1	General failure
-2	Invalid arguments
-3	O/S error
-4	Service not supported
-5	Underflow (number too small)
-6	Overflow (number too large)
-7	Out of range
-8	Divide by zero
-9	In use (e.g. serial port being used by another program)
-10	No system memory
-11	Segment table full
-12	Semaphore table full
-13	Process table full/Too many processes
-14	Resource already open
-15	Resource not open
-16	Invalid image/device file
-17	No receiver
-18	Device table full
-19	File system not found (e.g. if you unplug cable to PC)
-20	Failed to start
-21	Font not loaded

-22	Too wide (dialogs)
-23	Too many items (dialogs)
-24	Batteries too low for digital audio
-25	Batteries too low to write to Flash

FILE AND DEVICE ERRORS

-32	File already exists
-33	File does not exist
-34	Write failed
-35	Read failed
-36	End of file (when you try to read past end of file)
-37	Disk full
-38	Invalid name
-39	Access denied (e.g. to a protected file on PC)
-40	File or device in use
-41	Device does not exist
-42	Directory does not exist
-43	Record too large
-44	Read only file
-45	Invalid I/O request
-46	I/O operation pending
-47	Invalid volume (corrupt disk)
-48	I/O cancelled
-50	Disconnected
-51	Connected
-52	Too many retries
-53	Line failure
-54	Inactivity timeout
-55	Incorrect parity
-56	Serial frame (usually because Baud setting is wrong)
-57	Serial overrun (usually because Handshaking is wrong)
-58	Cannot connect to remote modem
-59	Remote modem busy
-60	No answer from remote modem

-61	Number is black listed (you may try a number only a certain number of times; wait a while and try again)
-62	Not ready
-63	Unknown media (corrupt SSD)
-64	Root directory full (on any device, the root directory has a maximum amount of memory allocated to it)
-65	Write protected
-66	File is corrupt (Media is corrupt on Series 3c)
-67	User abandoned
-68	Erase pack failure
-69	Wrong file type

TRANSLATOR ERRORS

-70	Missing “
-71	String too long
-72	Unexpected name
-73	Name too long
-74	Logical device must be A-Z (A-D on Series 5)
-75	Bad field name
-76	Bad number
-77	Syntax error
-78	Illegal character
-79	Function argument error
-80	Type mismatch
-81	Missing label
-82	Duplicate name
-83	Declaration error
-84	Bad array size
-85	Structure fault
-86	Missing endp
-87	Syntax Error
-88	Mismatched (or)
-89	Bad field list
-90	Too complex
-91	Missing ,

-92	Variables too large
-93	Bad assignment
-94	Bad array index
-95	Inconsistent procedure arguments

OPL SPECIFIC ERRORS

-96	Illegal Opcode (corrupt module translate again)
-97	Wrong number of arguments (to a function or parameters to a procedure)
-98	Undefined externals (a variable has been encountered which hasn’t been declared)
-99	Procedure not found
-100	Field not found
-101	File already open
-102	File not open
-103	Record too big (data file contains record too big for OPL)
-104	Module already loaded (when trying to LOADM)
-105	Maximum modules loaded (when trying to LOADM)
-106	Module does not exist (when trying to LOADM)
-107	Incompatible translator version (OPL file needs retranslation)
-108	Module not loaded (when trying to UNLOADM)
-109	Bad file type (data file header wrong or corrupt)
-110	Type violation (passing wrong type to parameter)
-111	Subscript or dimension error (out of range in array)
-112	String too long
-113	Device already open (when trying to LOPEN)
-114	Escape key pressed
-115	Incompatible runtime version
-116	ODB file(s) not closed
-117	Maximum drawables open (maximum 8 windows and/or bitmaps allowed)
-118	Drawable not open
-119	Invalid Window (window operation attempted on a bitmap)
-120	Screen access denied (when run from Calculator)

⑤ SERIES 5 SPECIFIC ERRORS

-121	OPX not found
-122	Incompatible OPX version
-123	OPX procedure not found
-124	STOP used in callback from OPX
-125	Incompatible update mode
-126	In database transaction or started changing fields

❺ Constants for all error 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.

INDEX

OPL

ADVANCED TOPICS

Many of the subjects covered in this document may provide benefits for all levels of programmers.

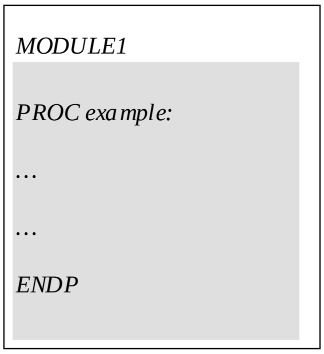
However, the subjects become progressively more technical. This User Guide cannot cover every OPL keyword in detail, as some give access to the innermost workings of the Psion. Some of these keywords are touched on in this section.

CONTENTS

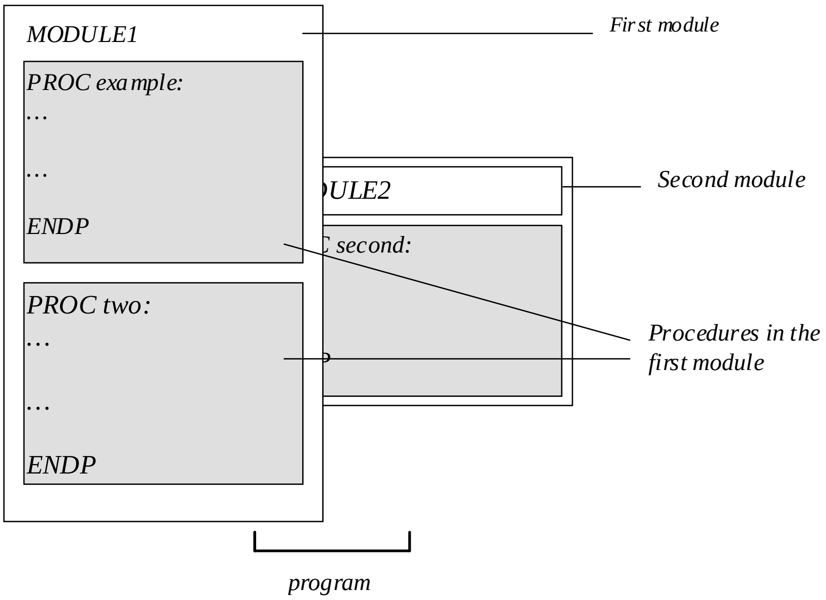
PROGRAMS, MODULES, PROCEDURES

PROGRAMS USING MORE THAN ONE MODULE

Program is the more general word for the “finished product” - a set of procedures which can be run. A program may consist of one module containing one or more procedures:



It may also consist of a number of modules containing various procedures:



A procedure in one module can call procedures in another module, provided this module has been loaded with the LOADM command. LOADM needs the filename of the module to load.

For example, if an OPL module `MAIN` has these two procedures:

```
PROC main:
   LOADM “library”
   pthis:                    REM run pthis: in this module
   pother:                   REM run pother: in “library”
   PRINT “Finished”
```

```
  PAUSE 40
  UNLOADM “library”
ENDP
```

```
PROC pthis:
  PRINT “Running pthis:”
  PRINT “in this module”
  PAUSE 40
ENDP
```

and a module called `LIBRARY` has this one:

```
PROC pother:
  PRINT “Running pother:”
  PRINT “in module called LIBRARY”
  PAUSE 40
ENDP
```

then running `MAIN` would display `Running pthis:` and `in this module`; then display `Running pother:` and `in module called LIBRARY`; and then display `Finished`.

You would usually only need to load other modules when developing large OPL programs.

❺ You can use LOADM up to seven times, to load up to seven modules.

❸ You can use LOADM up to three times, to load up to three modules.

If you want to load any further modules, you must first unload one of those currently loaded, by using UNLOADM with the filename. To keep your program tidy, you should UNLOADM a module as soon as you have finished using it.

If there is an error in the running program, you will only be positioned in the Program editor at the place where the error occurred if you were editing the module concerned (and you ran the program from there).

In spite of its name, LOADM does not “load” the whole of another module into memory. Instead, it just loads a block of data stored in the module which lists the names of the procedures which it contains. If such a procedure is then called, the procedure will be searched for, copied from file into memory and the procedure will then be run.

The modules are searched through in the order that they were loaded, with the module loaded last searched through last. **You may make considerable improvements in speed if you keep as few modules as possible loaded at a time (so avoiding a long search for each procedure) and if you load the modules with the most commonly called procedures first.**

CACHEING PROCEDURES FOR SPEED

❺ **On the Series 5, procedures are automatically cached.**

❸ On the Series 3c, without procedure cacheing, procedures are loaded from file whenever they are called, and discarded when they return. This is true even when procedures are all in one module. (With more than one module, LOADM simply loads a map listing procedure names and their positions in the module file so that they can be loaded fairly efficiently. It does not load procedures into memory.)

If a well-designed program calls procedures regularly, you can speed it up by cacheing procedures. This keeps the code for a procedure loaded in memory after it returns, so that if it is called again there is no need to fetch it from file again. The CACHE command takes two arguments the initial size of the cache and the maximum size (both in bytes). These can be up to 32,767 bytes. The minimum in both cases is 2000 bytes.

For a small program, you might use `CACHE 2000,2000` at the start of the program. Up to 2000 bytes of procedure code will be cached. If the cache fills up, and a procedure is called which is not in the cache, space will be made for it in the cache by removing other procedures from it.

For a much larger program, you might use `CACHE 10000,10000`. You may wish to change the settings and find the smallest setting which produces the maximum speed improvement.

Once a cache has been created, `CACHE OFF` prevents further cacheing, although the cache is still searched when calling subsequent procedures. `CACHE ON` may then be used to re-enable cacheing.

CALLING PROCEDURES BY STRINGS

Procedures can be called using a string expression for the procedure name. Use an `@` symbol, optionally followed by a character to show what type of value is returned for example, `%` for an integer. Follow this with the string expression in brackets. You can use upper or lower case characters.

Here are examples showing the four types of value which can be returned:

`i% = @%(name$):(parameters)`	for integer
`l& = @&(name$):(parameters)`	for long integer
`s$ = @$(name$):(parameters)`	for string
`f  = @(name$):(parameters)`	for floating-point

So, for example, `test$:(1.0,2)` and `@$("test"):(1.0,2)` are equivalent.

Note that the string expression does not itself include a symbol for the type (`%`, `&` or `$`).

You may find this useful if, in a more complex program, you want to “work out” the name of a procedure to call at a particular point. The section on ‘Friendlier interaction’ in the ‘GUI.pdf’ document includes an example program which uses this method.

WHERE FILES ARE KEPT

On the Series 3c the Siena and the Series 5, the internal memory and disks (memory disks on the Series 5; SSDs on the Series 3c) use a DOS-compatible directory structure, the same as that used on disks on PCs. However, the two machines differ quite considerably in this area as described below.

WHERE FILES ARE KEPT ON THE SERIES 5

To specify a file completely, the Series 5 uses a *drive* (or *device*), *folder* and *filename*:

- The drive is the area on the Series 5 where the file is kept. This can be C: (internal disk), D: (memory disk drive) E:,…, Y: or Z: (ROM).

- Every drive has one *root folder*, usually written as a backslash (\). This can “contain” files and/or other folders, each of which can contain more files and/or more folders. When you have “folders in folders” like this, they’re often called *subfolders.* Their names show where they are. For example, the root folder (\) could contain a folder called \JO, which might in turn contain a folder called \JO\BACKUP, which might contain some files.

- Filenames are composed of up to 256 characters (but see below), optionally followed by a *file extension* consisting of a dot and from one to three characters. A filename may not begin with a back or forward slash (\ or /) or a colon (:). Any trailing zeros on the filename are stripped.
  The Series 5 filing system does not treat extensions as a special component of a filename except when parsing (i.e. for PARSE$) where the extension is treated as on the Series 3c (see below). So while `myfile.` on the Series 3c meant the same as `myfile`, on the Series 5 the dot is actually part of the name. You can also use long filenames with embedded spaces and any number of dots like: `OPL User Guide` or `Very.long.filename`.

To specify a file completely, you add the three parts together. The folder part must end with a backslash. So an OPL module named `TEST`, in a folder called `\JO` in the Series 5 internal memory can be specified as `C:\JO\TEST`. If this file were in the `\JO\BACKUP` folder, it would be completely specified as `C:\JO\BACKUP\TEST`. If it were in the root folder, you would specify it as `C:\TEST`.

A full file specification may be up to 255 characters long for OPL.

In OPL, as in other applications, the files are kept on the drive and in the folder you specify.

USING FILE SPECIFICATIONS IN OPL

OPL commands which specify a filename, such as OPEN, CREATE, gLOADBIT and so on, can also use any or all of the other parts that make up a full file specification. (Normally, the only other part you might use is the drive name, if the file were on a memory disk.) So for example, `OPEN “C:\ADDR.TXT”` tries to open a file called `ADDR.TXT` in the root folder of the internal disk.

You can use the PARSE$ function if you need to build up complex filenames. See the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document for more details of PARSE$.

CURRENT FOLDER

The current folder for all commands is always `C:\` unless it has been changed by the command SETPATH. Hence any use of a keyword which takes a filename as an argument will only look in the current folder and so if this is other than `C:\`, it should be specified either by SETPATH or by including it in the filename. For example, to check whether the file `Program1` in the directory `D:\MyPrograms\` exists, either

```
SETPATH “d:\MyPrograms\”
...
IF EXISTS (“Program1”)
...
```

or

```
IF EXISTS (“d:\MyPrograms\Program1”)
...
```

CONTROL OF FOLDERS

Use the MKDIR command to make a new folder. For example, `MKDIR “C:\MINE\TEMP”` creates a `C:\MINE\TEMP` folder, also creating `C:\MINE` if it is not already there. An error is raised if the chosen folder exists already. Use `TRAP MKDIR` to avoid this.

SETPATH sets the current folder for file access - for example, `SETPATH “C:\DOCUMENTS”`. LOADM continues to use the folder of the running program, but all other file access will be to the newly specified folder.

Use RMDIR to remove a folder - for example, `RMDIR “C:\MINE”` removes the `MINE` folder on `C:`. A ‘Does not exist’ error is raised if the folder does not exist. Use `TRAP RMDIR` to avoid this. A ‘*File* is in use’ error will result if you try to remove a folder which contains open files.

WHERE FILES ARE KEPT ON THE SERIES 3C AND SIENA

Note that any mention of SSDs in this section refers to the Series 3c only. The Siena does not have an SSD drive. Other discussion refers to both machines as usual.

The Series 3c hides the complexities of the directory structure. You have only to supply a filename, and to say whereabouts a file is stored - internally, on an SSD or on another computer to which the Series 3c is linked.

In fact, in order to specify a file completely, the Series 3c uses a *filing system*, *drive* (or *device*), *directory* and *filename*:

- The filing system usually specifies the computer, and is usually LOC:: (‘local’ the Series 3c) or REM:: (‘remote’ an attached computer). This is always three letters and two colons.

- The drive is the area on that computer where the file is kept. On the Series 3c this can be M: (internal disk), A: (left SSD drive) or B: (right SSD drive).

- Every drive has one *root directory*, usually written as a backslash (\). This can “contain” files and/or other directories, each of which can contain more files and/or more directories. When you have “directories in directories” like this, they’re often called *subdirectories*. Their names show where they are. For example, the root directory (\) could contain a directory called \JO, which might in turn contain a directory called \JO\BACKUP, which might contain some files.

- Filenames are composed of one to eight letters and/or numbers, optionally followed by a *file extension* comprised of a dot and from one to three letters/numbers. File extensions are by convention used to group different types of files. The Series 3c uses file extensions in this way, but hides this from you.

To specify a file completely, you add the four parts together. The directory part must end with a backslash. So an OPL module named `TEST`, in a directory called `\JO` in the Series 3c internal memory can be specified as `LOC::M:\JO\TEST.OPL`. If this file were in the `\JO\BACKUP` directory, it would be completely specified as `LOC::M:\JO\BACKUP\TEST.OPL`. If it were in the root directory, you would specify it as `LOC::M:\TEST.OPL`.

A full file specification may be up to 128 characters long.

In OPL, unless you say otherwise, files are kept on the Series 3c (`LOC::`), in the internal memory (`M:`). The directories and file extensions used are:

Type of file	*Directory*	*File extension*
OPL modules	\OPL	.OPL
translated modules	\OPO	.OPO
data files	\OPD	.ODB
bitmaps	\OPD	.PIC

USING FILE SPECIFICATIONS IN OPL

OPL commands which specify a filename, such as OPEN, CREATE, gLOADBIT and so on, can also use any or all of the other parts that make up a full file specification. (Normally, the only other part you might use is the drive name, if the file were on an SSD.) So for example, `OPEN “REM::C:\ADDR.TXT”`... tries to open a data file called `ADDR.TXT` on the root directory of a hard disk `C:` on an attached PC.

You can use the PARSE$ function if you need to build up complex filenames. See the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document for more details of PARSE$.

Unless you have a good reason, though, it’s best not to change directories or file extensions for files on the Series 3c. You can lose information this way, unless you’re careful.

CONTROL OF DIRECTORIES

Use the MKDIR command to make a new directory. For example, `MKDIR “M:\MINE\TEMP”` creates a `M:\MINE\TEMP` directory, also creating `M:\MINE` if it is not already there. An error is raised if the chosen directory exists already - use `TRAP MKDIR` to avoid this.

SETPATH sets the current directory for file access - for example, `SETPATH “A:\DOCS”`. LOADM continues to use the directory of the running program, but all other file access will be to the new directory instead of `\OPD`.

Use RMDIR to remove a directory - for example, `RMDIR “M:\MINE”` removes the `MINE` directory on `M:\`. An error is raised if the directory does not exist use `TRAP RMDIR` to avoid this.

You can only remove empty directories.

FILE SPECIFICATIONS ON REM::

You should not assume that remote file systems use DOS-like file specifications for example, an Apple Macintosh specification is `disk:folder:folder:filename`. You can only assume that there will be four parts - disk/device, path, filename and (possibly null) extension. PARSE$, however, will always build up or break down REM:: file specifications correctly.

SAFE POINTER ARITHMETIC

❺ Note that on the Series 5, if you are using 32-bit addressing, as will be the case by default (see the ‘32-bit addressing’ section later in this document), you should use ordinary long integer arithmetic and should **not** use UADD and USUB.

However, **on the Series 3c and Siena or if you have used SETFLAGS on the Series 5 to enforce the 64K memory limit**, whenever you wish to add or subtract something from an integer which is an address of something (or a pointer to something) you should use UADD and USUB. If you do not, you will risk an ‘Integer overflow’ error.

An address can be any value from 0 to 65535. An integer variable can hold the full range of addresses, but treats those values from 32768 to 65535 as if they were -32768 to -1. If your addition or subtraction would convert an address in the 0-32767 range to one in the 32768-65535 range, or vice versa, this would cause an ‘Integer overflow’.

UADD and USUB treat integers as if they were unsigned, i.e. as if they really held a value from 0 to 65535.

For example, to add 1 to the address of a text string (in order to skip over its leading byte count and point to the first character in the string), use `i%=UADD(ADDR(a$),1)`, not `i%=ADDR(a$)+1`.

USUB works in a similar way, subtracting the second integer value from the first integer, in unsigned fashion for example, `USUB(ADDR(c%),3)`.

✎ `USUB(x%,y%)` has the same effect as `UADD(x%,-y%)`.

OPL APPLICATIONS (OPAS)

You can make an OPL program appear as an icon in the system screen, and behave like the other icons there, by converting it into an *OPL application*, or *OPA*. The support for OPAs on the Series 5 is changed and extended from that of the Series 3c and Siena. The following two sections deal separately with these systems.

OPAS ON THE SERIES 5

There are two settings for OPAs which are set using the FLAGS command (similar to TYPE on the Series 3c):

- FLAGS 1 is used if your application can create files. It will then be included in the list of applications offered when the user creates a new file from the System screen (like Program, Word etc.).

- FLAGS 2 prevents the application from appearing in the Extras bar in the System screen. It is very unusual to have this flag set.

Once created, OPAs may be used in the same way as the built in Series 5 applications: new document files may be created from the System screen using ‘New File’ if the flag 1 is set and/or from the Extras bar if the flag 2 is **not** set. Existing files may be opened by selecting them on the System screen as usual. You can stop a running OPA by using the Task list.

DEFINING AN OPA

To make an OPA, your OPL file should **begin with** the APP keyword, followed by a name for the OPA in the machine’s default language and its UID. The name may be up to 250 characters. (Note that it does not have quote marks.) If the CAPTION command is used, however, this default name will be discarded (see below).

The UID specifies the UID of the application. For applications that are to be distributed, UIDs are reserved by contacting Psion. These official UIDs are guaranteed to be unique to the application and have values of &10000000 and above. To obtain a reserved UID you should contact Psion Software in one of the following ways:

e-mail to **uid_sw@software.psion.com**

or use the EPOC World web site, **http://www.software.psion.com/EPOCWorld/**

or fax to **+44-171-724-4048, attn UID Allocations**

or write to **UID Allocations, Psion Software plc, 19 Harcourt St, London W1H 1DT, England**

For applications developed for personal use there is no need to reserve official UIDs, and any UID between &01000000 and &0fffffff may be selected. However, if the UIDs of two different applications did clash, then either your application’s documents will have the wrong icon and selecting these will run the other application, or the other application’s documents will seem to belong to your application.

The APP line may be followed by any or all of the keywords CAPTION, ICON and FLAGS. Finally, use ENDA, and then the first procedure may begin as in a normal OPL file. Here is an example of how an OPA might start:

```
APP Expenses,&20000000       REM !!! Should be a reserved UID !!!
   FLAGS 1
   ICON “icon.mbm”
   CAPTION “Expenses”,1      REM English name
   CAPTION “Expenses”,10     REM American name
ENDA
```

Here is another example:

```
APP Picture,&30000000         REM !!! Should be a reserved UID !!!
   FLAGS 2
   ICON “picicon.mbm”
ENDA
```

FLAGS takes an integer argument of 1 or 2. The meanings of these are outlined above. If you don’t specify the flags, none are set.

ICON gives the name of a *multi-bitmap file* (MBM), also known as an *EPOC Picture file*, which contains the icons for an OPL application. These icons are used on the Extras bar and for the application’s documents on the System screen. The multi-bitmap file can contain up to three bitmaps of sizes 24×24, 32×32 and 48×48 pixels respectively, **but each must be paired with a mask**. Each bitmap should be followed by its mask in the multi-bitmap file, so that bitmaps and masks occur alternately. The pixels which are set in the mask specify pixels in the bitmap which are to be used. Pixels which are clear in the mask specify pixels that are not to be used from the bitmap, allowing the background to be displayed in these pixels. Only pixels which are set in the mask are drawn in the final icon.

The different sizes of bitmaps are used for the different zoom levels in the System screen. The sizes are read from the MBM and the most suitable size is zoomed if the exact sizes required are not provided or if some are missing.

You can use ICON more than once within the APP...ENDA construct. The translator only insists that all icons are paired with a mask of the same size in the final ICON list. This allows you to use MBMs containing just one bitmap, as produced by the Sketch application. Icons may also be created on a PC (if the appropriate tools are available), or of course by another OPL program or application.

CAPTION specifies an application’s *public name* (or *caption*) for a particular language, which is the name which will appear below its icon on the Extras bar and in the list of ‘Programs’ in the ‘New File’ dialog (assuming the setting of FLAGS allows these) when that is the language used by the machine. This name is also used as the default document name for documents launched using the Extras bar. The maximum length of the caption is 255 characters. However, note that a caption no longer than around 8 letters will look best on the Extras bar. If any CAPTION statement is included in the APP…ENDA structure, then the default caption provided by the APP declaration will be discarded. Therefore as many statements of CAPTION as are necessary to cover all the languages required, including the language of the machine on which the application is originally developed, should be used. The constants for the language codes 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.

On creation of an application, a folder specifically for the application is also created. For example, if the OPA has the name `AppXxx` the folder created will be `\System\Apps\AppXxx\`. The APP file itself (the translated OPL module) and the *application information file* (AIF) which contains three icon and mask pairs and the application caption and flags, are created in this folder. For example, the AIF for `AppXxx.app` would be `\System\Apps\AppXxx\AppXxx.aif`. The AIF file will be generated based on all the information contained in the APP…ENDA construct, but if any information is missing defaults will be used. These are:

- for ICON: the default (question mark) icons
- for CAPTION: the caption specified in the APP declaration
- for FLAGS: the default value of 0.

✎ The arguments to any of the keywords between APP and ENDA must be constants and not expressions.

RUNNING THE OPA

Once you’ve successfully translated the OPL file, the applications icon will automatically appear on the Extras bar and/or on the ‘Program’ list in the ‘New File’ dialog (as specified by the FLAGS setting). The new documents of the application may then be created and existing ones opened as with other built in Series 5 applications.

The first thing a file-based OPA should do is to get the name of the file to use, and check whether it is meant to create it or open it. `CMD$(2)` returns the full name of the file to use; `CMD$(3)` returns “C” for “Create”, “O” for “Open” or “R” for “Run”.

All document-based OPAs should handle all of these cases; for example, if a “Create” fails because the file exists already, or an “Open” fails because it does not, OPL raises the error, and the OPA should take suitable action. Note however, that in general the System screen will not allow such events to occur and therefore they are unlikely to happen.

“R” means that your application has been run from the OPL Program editor or has been selected via the application’s icon on the Extras bar, and not by the selection or creation of one of its documents from the system screen. A default filename, including path, is passed in `CMD$(2)`. When “R” is passed, an application should always try to open the last-used document. This is the document that was in use when the application was last closed, **not** the document that was most recently opened. The name of this document should be stored in a `.INI` file with the same name and in the same folder as your application. So for example, `AppXxx.app` would have `.INI` file `\System\Apps\AppXxx\AppXxx.ini`. If the `.INI` file does not exist or cannot be opened for any reason, or if the document listed there no longer exists, you should create the document named in `CMD$(2)`. `CMD$(2)` is a default name provided by the System based on your application’s caption. If the `.INI` file is corrupt it should be deleted before going on to create `CMD$(2)`. No error message should be displayed in this case.

Constants for the array indices of CMD$ and the return values of `CMD$(3)` 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.

HOW THE SERIES 5 TALKS TO AN OPA

When the Series 5 wants an OPA to exit or to switch files, it sends it a *System message*, in the form of an event. For example, this would happen if the ‘Close file’ option in the Task list were used to stop a running OPA.

TESTEVENT and GETEVENT32 (synchronous) and GETEVENTA32 (asynchronous) check for certain events, including both keypresses and System messages. All types of OPA **must** use these keywords to check for **both** keypresses and System messages; keyboard commands such as GET, KEY and KEYA cause other events to be **discarded**.

GETEVENT32 waits for an event whereas TESTEVENT simply checks whether an event has occurred without getting it. If TESTEVENT returns non-zero, an event has occurred, and can be read with GETEVENT32. However, it is recommended that you use either GETEVENT32 or GETEVENTA32 without TESTEVENT as **TESTEVENT may use a lot of power, especially when used in a loop** as will often be the case. GETEVENT32 may be used if you only wish to pick up window server events, but otherwise you should use the asynchronous GETEVENTA32. TESTEVENT should generally only be used when polling while doing something else in background.

GETEVENT32 and GETEVENTA32 both take one argument: the name of a long integer array, for example, `GETEVENT32 ev&()`. The array should contain at least 16 long integers.

For example, if the event is a keypress`(ev&(1) AND &400) = 0` and,

`ev&(1)` = keycode (as for GET)

`ev&(2)` = time stamp (gives the time of the keypress) `ev&(3)` = scan code (locates the key on the keyboard)

`ev&(4)` = modifier code (e.g. Shift, Control)

`ev&(5)` = repeat. Note that this is strictly the repeat value, i.e. if there is only one keypress, then the value of `ev&(5)` is 0.

For non-key events `(ev&(1) AND &400)` will be non-zero. On the Series 5, these include *pointer events* (pen events). For an application, it may be suitable to filter out certain pointer events. This may be done using the POINTERFILTER command. POINTERFILTER takes two arguments: a filter and a mask. Each of the bits in these two arguments represents a certain pointer event, allowing a flag to be set to say whether that event should be filtered out. The bits which are set in the mask specify bits in the filter which are to be used. Bits which are clear in the mask specify bits that are not to be used from the filter. This makes it possible to filter out some pointer events, filter back in some others and leave the settings of some alone all in one call to POINTERFILTER. To set or clear flags you would set and clear them in the filter and set all the flags that require changing in the mask. To leave some bits in the filter alone just don't set the bits in the mask. See the 'Alphabetic Listing' section of the 'Glossary.pdf' document for more details of POINTERFILTER and the values returned to `ev&()` by GETEVENT32 and GETEVENTA32.

If the event is a System message to change files or quit, `ev&(1)=&404`. You should then use GETCMD$ to find the action required.

GETCMD$ returns a string, whose first character is "C", "O" or "X".

You can only call GETCMD$ once for each event. You should do so as soon as possible after reading the event. Assign the value returned by GETCMD$ to a string variable so that you can extract its components.

If you have `c$=GETCMD$`, the first character, which you can extract with `LEFT$(c$,1)`, has the following meaning:

"C" - close the current document, and create the specified new file.

"O" - close the current document, and open the specified existing file.

"X" - save the name of the current document (if any) to the `.INI` file, close the current document and quit the OPA.

Again with `c$=GETCMD$`, `MID$(c$,2,255)` is the easiest way to extract the filename.

✎ Note that events are ignored while you are using keywords which pause the execution of the program GET, GET$, EDIT, INPUT, PAUSE, MENU and DIALOG. If you need to use these keywords, use `LOCK ON` / `LOCK OFF` (described later) around them to prevent the System screen from sending messages.

SYSTEM SCREEN COMPLIANCE

A well-behaved Series 5 application should obey the following important guidelines:

- All applications should respond in the standard way to the initial command-line and to any System screen messages to switch files. See the 'How the Series 5 talks to an OPA' section above.
- All applications should support a toolbar. Toolbar.opo is supplied in the ROM and provides the set of procedures required for this support. See the 'Friendlier Interaction' section of the 'GUI.pdf' document.
- All Series 5 applications should save non-document files (external files) to their so-called *application directory*. For an application called AppXxx, the application directory is \System\Apps\AppXxx\. This is also the directory where the application itself is saved.
  You can find out the full file specification of external files by using PARSE$. For example,
  `p$=PARSE$("NEW",LEFT$(CMD$(1),LEN(CMD$(1)-4),x%())`

`CMD$(1)` gives the device and the path of the OPL application and the name of the file.
By default the system screen hides the System folder and its subfolders. Use the ‘Preferences’ option in the ‘Tools’ menu (Ctrl+K) in the System screen and check the checkboxes for both ‘Show hidden files’ and ‘Show ‘System’ folder’ to show all applications and their files.

- The FLAGS keyword in the APP...ENDA construct should have the value 1 if the application supports *documents* (a file that will run your application when selected from the System screen). Without this flag the application will not be listed when the user chooses to create a new file from the System screen (see sections above).
- You should set ESCAPE OFF so that it is impossible for the user to stop the application running by hitting Ctrl+Esc.

LAUNCHING HELP FILES

It is possible to write your own Help for your own application, which should be created in the Data application. The file should be similar to the built-in help, using two labels only for each entry to give a title and the explanation itself. The file should be stored in the application’s folder and must have the filename extension `.hlp`. The procedure RUNAPP&: in System OPX (see the ‘OPX.pdf’ document for more details) may then be used to launch Data and open the application help file. For example, if you want to run a help file called `Myapp.hlp`, the help file for your own application called `Myapp`, you should use:

```
drv$=LEFT$(CMD$(1),2)
```

```
RUNAPP&:(“Data”,drv$+“\System\Apps\Myapp\Myapp.hlp”,“”,0)
```

You should make it possible for users of your application to launch your help file from the application’s menus. Following the Series 5 style guide, the ‘Help’ item should appear on the ‘Tools’ menu at the bottom of the section above ‘Infrared’, so it will be followed by a separating line. The shortcut key should be Ctrl+Shift+H.

The example below shows how to bring the custom Help to foreground when the user chooses ‘Go back’ and then chooses custom Help again, catering for the possibility that the user may also have exited Help. The example also shows how an application should end the custom help task, if any, on exit.

If custom help has been launched, the global HelpThread& will be non-zero. It is possible though that the user has exited custom Help either before or after returning to the application, without the application’s knowledge. In this case SETFOREGROUNDBYTHREAD&:(HelpThread&,0) will raise an error because the thread doesn’t exist. ONERR noHelpThread will then cause control to pass to the RUNAPP&: to start a new Help thread running.

Similarly, calling ENDTASK&:(HelpThread&,0) raises an error if the thread no longer exists, so error handling should be used in this case simply to ignore the error.

SETFOREGROUNDBYTHREAD&: and ENDTASK&: are supplied by System.opx as declared in System.oxh

```
INCLUDE “System.oxh”
```

```
PROC Help:
  IF EXIST (Helpfile$)
    IF HelpThread&<>0
        ONERR noHelpThread      REM go to noHelpThread if user exited Help
        SETFOREGROUNDBYTHREAD&:(HelpThread&,0)
    ELSE
  noHelpThread::
        ONERR OFF
```

```
         HelpThread&=RUNAPP&:("Data",Helpfile$,"",0)
     ENDIF
   ENDIF
ENDP
```

and in

```
PROC Exit:
   IF HelpThread&<>0
     ONERR noHelpThread            REM go to noHelpThread if user exited Help
     ENDTASK&:(HelpThread&,0)
   ENDIF
noHelpThread::
   STOP
ENDP
```

EXAMPLE OPAS

Here is an OPA which just prints the keys you press. It is not a document-based application (like Calc, but unlike Word which is document-based) so it uses `FLAGS 0`. The keyboard procedure `getk&:` returns the key pressed, as with GET, but jumps to a procedure `endit:` if a System message to close down is received. (OPAs with flags set to 0 do not receive "change file" messages.)

```
CONST KUidAppMyApp0&=&40000000   REM !!! Should be a reserved UID!!!
APP myapp0,KUidAppMyApp0&
   CAPTION "Get Key",1
   ICON "myapp0.mbm"
ENDA

PROC start:
   GLOBAL a&(10),k&
   FONT 11,16
   PRINT "Q to Quit"
   PRINT " or select 'Close file'"
   PRINT " from the Task List"
   DO
     k&=getk&:
     PRINT CHR$(k&);
   UNTIL (k& AND &ffdf)=%Q        REM Quick way to do uppercase
ENDP

PROC getk&:
   DO
     GETEVENT32 a&()
     IF a&(1)=&404
          IF LEFT$(GETCMD$,1)="X"
               endit:
          ENDIF
     ENDIF
   UNTIL a&(1)<256
   RETURN a&(1)
ENDP
```

```
PROC endit:
   STOP
ENDP
```

Here is a similar document-based OPA. It does the same as the previous example, but System messages to change files cause the procedure `fset:` to be called. The relevant files are opened or created. A proper application with a toolbar would call `TBarSetTitle:` to change its title. See the ‘Friendlier Interaction’ section of the ‘GUI.pdf’ document.

```
CONST KUidAppMyApp1&=&50000000   REM !!! Should be a reserved UID!!!
APP myapp1,KUidAppMyApp1&
   FLAGS 1
   CAPTION “Get Key Doc”,1
   ICON “myapp1.mbm”
ENDA

PROC start:
   GLOBAL a&(10),k&,w$(255)
   FONT 11,16 :w$=CMD$(2)
   fset:(CMD$(3))
   PRINT “Q to Quit”
   PRINT “use the Task list”
   PRINT “to create/switch files”
   DO
      k&=getk&:
      PRINT CHR$(k&);
   UNTIL (k& AND &ffdf)=%Q
ENDP

PROC getk&:
   LOCAL t$(1)
   DO
      GETEVENT32 a&()
      IF a&(1)=$404
            w$=GETCMD$
            t$=LEFT$(w$,1)
            w$=MID$(w$,2,255)
            IF t$=“X”
                  endit:
            ELSEIF t$=“C” OR t$=“O”
                  TRAP CLOSE
                  IF ERR
                        GIPRINT ERR$(ERR)
                        CONTINUE
                  ENDIF
                  fset:(t$)
            ENDIF
      ENDIF
   UNTIL a&(1)<256
   RETURN a&(1)
ENDP
```

```
PROC fset:(t$)
  LOCAL p&(6)
  IF t$=“C”
    TRAP DELETE w$
    SETDOC w$
    TRAP CREATE w$,A,A$
  ELSEIF t$=“O”
    SETDOC w$
    TRAP OPEN w$,A,A$
  ENDIF
  IF ERR
    CLS :PRINT ERR$(ERR)    REM should revert to old file if possible
    GET :STOP
  ENDIF
ENDP

PROC endit:
  STOP
ENDP
```

You should, as in both these examples, be precise in checking for the System message; if in future the GETCMD$ function were to use values other than “C”, “O” or “X”, these procedures would ignore them.

If you need to check the modifier keys for the returned keypress, use `a&(4)` instead of KMOD.

SETDOC called just before the creation of the file ensures that the created file is a *document*, i.e. that the file launches its associated application when selected. The strings passed to SETDOC and CREATE (or gSAVEBIT) should be exactly the same, otherwise a non-document file will be created. SETDOC should also be called when opening a document to allow the System screen to display the correct document name in its task list. OPL explicitly supports database and multi-bitmap files as documents.

To be strict, whenever **creating** a file, an OPA should first use PARSE$ to find the disk and directory requested. It should then use `TRAP MKDIR` to ensure that the directory exists.

WHEN AN OPA CANNOT RESPOND

The LOCK command marks an OPA as locked or unlocked. When an OPA is locked with `LOCK ON`, the System will not send it events to change files or quit. If, for example, you attempt to close down the OPA from the Task list, a message will appear, indicating that the OPA cannot close down at that moment.

You should use `LOCK ON` if your OPA uses a keyword, such as MENU, DIALOG and EDIT, which pauses the execution of the program. You might also use it when the OPA is about to go busy for a considerable length of time, or at any other point where a clean exit is not possible. Do not forget to use `LOCK OFF` as soon as possible afterwards.

An OPA is initially unlocked.

OPAS ON THE SERIES 3C AND SIENA

There are five different types of OPA, called type 0 to type 4:

- TYPE 0 (like Calc): The OPA uses no files.
- TYPE 1: Only one file is used. A type 1 OPA will look the same as a type 0. The only difference is that the type 1 is using a file, of the same name as the OPA.
- TYPE 2 (like World): You can have more than one file, but only one can be in use (bold) at any time. When you pick a new file to use, its name becomes bold, and the one that was previously bold reverts to normal. **What has actually happened is that the running OPA has switched files** - it has **not** closed down, and no new copy of the OPA is run.
- TYPE 3 (like Data, Word, Agenda, Sheet): You can have more than one file, and any number may be open (bold) at a given time.
  When you select a new file, one of the running OPAs normally switches to this file, as with type 2 OPAs. You can, however, with Shift-Enter, start a new OPA running just for this file, without a different file exiting.
- TYPE 4 (like RunOpl): Many files can be used, and any number may be in use at a given time. When you select a new file, a new version of the OPA is always run, to use the new file.

Types 3 and 4 allow more than one file to be **in use** (i.e. have their names in bold). When this happens **a separate version of the OPA runs for each bold file**. With types 0, 1 and 2, only one version of the OPA can be running at any time.

Initially, the OPA's name appears beneath the icon. If you move onto this name and press Enter, file-based OPAs (types 1 to 4) will use a file of this name. Types 2, 3 and 4 allow you to create lists of files below the icon (with the 'New file' option). You use the file lists in the same way as the lists under the other icons in the System screen.

You can stop a running OPA by moving the cursor onto its bold name and pressing Delete. After a 'Confirm' dialog, the System screen tells the OPA to stop running.

DEFINING AN OPA

To make an OPA, your OPL file should **begin with** the APP keyword, followed by a name for the OPA. The name should begin with a letter, and comprise of 1 to 8 letters and/or numbers. (Note that it does not have quote marks.) The APP line may be followed by any or all of the keywords PATH, EXT, ICON and TYPE. **A Series 3c OPA should also add $1000 to the type if it has its own 48x48 pixel, black/grey icon** (see the discussion of ICON below for details). Finally, use ENDA, and then the first procedure may begin as in a normal OPL file. Here is an example of how an OPA might start:

```
APP Expenses
  TYPE $1003
  PATH “\EXP”
  EXT “EXP”
  ICON “\OPD\EXPENSES.PIC”
ENDA
```

Here is another example:

```
APP Picture
   TYPE 1
ENDA
```

TYPE takes an integer argument from 0 to 4. The various types of OPA are outlined earlier. If you don't specify the type, 0 is used.

PATH gives the directory to use for this OPA's files. If you do not use this, the normal `\OPD` directory will be used. The maximum length, including a final `\`, is 19 characters. Don't include any drive name in this path.

EXT gives the file extension of files used by this OPA. If you do not specify this, `.ODB` is used. Note that the files used by an OPA do not have to be data files, as the I/O commands give access to files of all kinds. EXT does not define the file type, just the file extension to use. However, **for simplicity's sake, examples in this section use data files**.

(PATH and EXT provide information for the System screen - they do not affect the program itself. The System screen displays under the OPA icon all files with the specified extension in the path you have requested.)

ICON gives the name of the bitmap file to use as the icon for this OPA. If no file extension is given, .PIC is used. If you do not use ICON, the OPA is shown on the System screen with a standard OPA icon.

As mentioned above, you should add $1000 to the argument to TYPE for a Series 3c icon. This specifies that the icon has size 48x48 pixels (instead of 24x24 as it was on the Series 3). If the first bitmap has size 24x24, it is ignored and the following two bitmaps must be the 48x48 black and grey icons respectively. If the first bitmap is 48x48, it is assumed to be the black icon and the grey icon must follow. **If $1000 is not set, a scaled up 24x24 icon will be used.** The translator does not check the size of the icons. If you want to design your own icon using an OPL program, see gSAVEBIT for details on saving both black and grey planes to a bitmap file.

The arguments to any of the keywords between APP and ENDA must be constants and not expressions. So, for example, you must use `TYPE $1003` instead of `TYPE $1000 OR 3`.

RUNNING THE OPA

Once you've translated the OPL file, return to the System screen and use Install on the App menu to install the OPA in the System screen. (You only need to do this once.) Once installed, file-based OPAs are shown with the list of available files, if any are found. Otherwise, the name used after the APP keyword appears below the icon.

Note that the translated OPA is saved in a `\APP` directory. If you previously translated the module without the APP...ENDA at the start, the old translated version will still be listed under the RunOpl icon, and should be deleted.

The first thing a file-based OPA should do is to get the name of the file to use, and check whether it is meant to create it or open it. `CMD$(2)` returns the full name of the file to use; `CMD$(3)` returns "C" for "Create" or "O" for "Open". **All** file-based OPAs (types 1 to 4) should handle both these cases; if a "Create" fails because the file exists already, or an "Open" fails because it does not, OPL raises the error, and the OPA should take suitable action - perhaps even just exiting.

HOW THE SERIES 3C TALKS TO AN OPA

When the Series 3c wants an OPA to exit or to switch files, it sends it a *System message*, in the form of an event. This would happen if you press Delete to stop a running OPA, or select a new file for a type 2 or 3 OPA.

TESTEVENT and GETEVENT check for certain events, including both keypresses and System messages. All types of OPA **must** use these keywords to check for **both** keypresses and System messages; keyboard commands such as GET, KEY and KEYA cause other events to be **discarded**.

GETEVENT waits for an event whereas TESTEVENT simply checks whether an event has occurred without getting it.

If TESTEVENT returns non-zero, an event has occurred, and can be read with GETEVENT. This takes one argument, the name of an integer array for example, `GETEVENT a%()`. The array should be at least 6 integers long. (This is to allow for future upgrades you only need use the first two integers.)

If the event is a keypress:

`a%(1)` = keycode (as for GET)

`a%(2) AND $00ff` = modifier (as for KMOD)

`a%(2)/256` = auto-repeat count (ignored by GET; you can ignore it too)

For non-key events `(a%(1) AND $400)` will be non-zero. If the event is a System message to change files or quit, `a%(1)=$404`. You should then use GETCMD$ to find the action required.

GETCMD$ returns a string, whose first character is “C”, “O” or “X”. If it is “C” or “O”, the rest of the string is a filename.

You can only call GETCMD$ once for each event. You should do so as soon as possible after reading the event. Assign the value returned by GETCMD$ to a string variable so that you can extract its components.

If you have `c$=GETCMD$`, the first character, which you can extract with `LEFT$(c$,1)`, has the following meaning:

“C” - close down the current file, and create the specified new file.

“O” - close down the current file, and open the specified existing file.

“X” - close down the current file (if any) and quit the OPA.

Again with `c$=GETCMD$`, `MID$(c$,2,128)` is the easiest way to extract the filename.

Note that events are ignored while you are using keywords which pause the execution of the program GET, GET$, EDIT, INPUT, PAUSE, MENU and DIALOG. If you need to use these keywords, use `LOCK ON` / `LOCK OFF` (described later) around them to prevent the System screen from sending messages.

EXAMPLE OPAS

Here is a type 0 OPA, which just prints the keys you press. The keyboard procedure `getk%:` returns the key pressed, as with GET, but jumps to a procedure `endit:` if a System message to close down is received. (Type 0 OPAs do not receive “change file” messages.)

`getk%:` does not return events with values 256 ($100) or above, as they are not simple keypresses. This includes the non-typing keys like Menu ($100-$1FF), hot-keys ($200-$3FF), and non-key events ($400 and above).

```
APP myapp0
   TYPE $1000
   ICON “\opd\me”
ENDA

PROC start:
   GLOBAL a%(6),k%
   STATUSWIN ON :FONT 11,16
   PRINT “Q to Quit”
   PRINT “ or press Delete in”
   PRINT “ the System screen”
   DO
      k%=getk%:
      PRINT CHR$(k%);
   UNTIL (k% AND $ffdf)=%Q         REM Quick way to do uppercase
ENDP

PROC getk%:
   DO
      GETEVENT a%()
      IF a%(1)=$404
            IF LEFT$(GETCMD$,1)=“X”
                  endit:
            ENDIF
      ENDIF
   UNTIL a%(1)<256
   RETURN a%(1)
ENDP

PROC endit:
   STOP
ENDP
```

Here is a similar type 3 OPA. It does the same as the previous example, but System messages to change files cause the procedure `fset:` to be called. The relevant files are opened or created; the name of the file in use is shown in the status window.

```
APP myapp3
   TYPE $1003
   ICON “\opd\me”
ENDA

PROC start:
   GLOBAL a%(6),k%,w$(128)
   STATUSWIN ON :FONT 11,16 :w$=CMD$(2)
```

```
  fset:(CMD$(3))
  PRINT “Q to Quit”
  PRINT “ or press Delete in”
  PRINT “the System screen”
  PRINT “ or create/swap files in”
  PRINT “the System screen”
  DO
     k%=getk%:
     PRINT CHR$(k%);
  UNTIL (k% AND $ffdf)=%Q
ENDP

PROC getk%:
  LOCAL t$(1)
  DO
     GETEVENT a%()
     IF a%(1)=$404
          w$=GETCMD$
          t$=LEFT$(w$,1)
          w$=MID$(w$,2,128)
          IF t$=“X”
               endit:
          ELSEIF t$=“C” OR t$=“O”
               TRAP CLOSE
               IF ERR
                    CLS :PRINT ERR$(ERR)
                    GET :CONTINUE
               ENDIF
               fset:(t$)
          ENDIF
     ENDIF
  UNTIL a%(1)<256
  RETURN a%(1)
ENDP

PROC fset:(t$)
  LOCAL p%(6)
  IF t$=“C”
     TRAP DELETE w$         REM SYS.SCREEN DOES ANY “OVERWRITE?”
     TRAP CREATE w$,A,A$
  ELSEIF t$=“O”
     TRAP OPEN w$,A,A$
  ENDIF
  IF ERR
     CLS :PRINT ERR$(ERR)
     GET :STOP
  ENDIF
  SETNAME w$
ENDP
```

```
PROC endit:
   STOP
ENDP
```

You should, as in both these examples, be precise in checking for the System message; if in future the GETCMD$ function were to use values other than “C”, “O” or “X”, these procedures would ignore them.

If you need to check the modifier keys for the returned keypress, use `a%(2) AND $00ff` instead of KMOD.

SETNAME extracts the main part of the filename from any file specification (even one that is not DOS-like), in the same way as PARSE$. Using SETNAME ensures that the correct name will be used in the file list in the System screen. If an OPA lets you change files with its own ‘Open file’ option, it should always use SETNAME to inform the System screen of the new file in use.

To be strict, whenever **creating** a file, an OPA should first use PARSE$ to find the disk and directory requested. It should then use `TRAP MKDIR` to ensure that the directory exists.

WHEN AN OPA CANNOT RESPOND

The LOCK command marks an OPA as locked or unlocked. When an OPA is locked with `LOCK ON`, the System will not send it events to change files or quit. If, for example, you move onto the file list in the System screen and press Delete to try to stop that running OPA, a message will appear, indicating that the OPA cannot close down at that moment.

You should use `LOCK ON` if your OPA uses a keyword, such as EDIT, which pauses the execution of the program. You might also use it when the OPA is about to go busy for a considerable length of time, or at any other point where a clean exit is not possible. Do not forget to use `LOCK OFF` as soon as possible afterwards.

An OPA is initially unlocked.

DESIGNING AN ICON

As discussed earlier, an OPA icon is black and grey and has size 48 by 48 pixels. The icon is stored as two 48x48 bitmaps, black followed by grey, in a bitmap file. Here is a simple example program which creates a suitable bitmap:

```
PROC myicon:
   gCREATE(0,0,48,48,1,1)
   gBORDER $200
   gAT 6,28
   gPRINT “me!”
   gSAVEBIT “me”
ENDP
```

Here the window is created with a grey plane (the sixth argument to gCREATE) gSAVEBIT automatically saves a window with both black and grey plane to a file in the required format.

In the OPA itself use the ICON keyword, as explained previously, to give the name of the bitmap file to use here, `ICON “\opd\me”`.

OPAS AND THE STATUS WINDOW

If you use `STATUSWIN ON,2` to display the status window, it shows the OPA's own icon and the name used with the APP keyword. `STATUSWIN ON,1` displays the smaller status window.

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

You can also display a list of modes/views for use with the diamond key with DIAMINIT and position the diamond indicator with `DIAMPOS`.

The name can be changed with the SETNAME command. In general, an OPA should use SETNAME whenever it changes files, or creates a new file.

OTHER TYPE OPTIONS

You can add any of these numbers to the value you use with TYPE:

- $8000 (-32768) stops the System screen's 'New file' option from working, as for the RunOpl icon (translated OPL modules).

- $4000 (16384) stops the System screen from closing the OPA, as for the Time icon. You should not use this without a **very** good reason.

- $100 (8192) causes the System screen to terminate the OPA (when Delete is pressed there) without sending a message to the OPA to quit ("X"), as for the RunOpl icon again. This should be used only for OPAs which have no data that could be lost by sudden termination.

For example, use `TYPE $8001` for a type 1 OPA having the first of the features above. (Note that `TYPE $8000+1` would fail to translate as the translator cannot evaluate expressions for any keywords between APP and ENDA).

TRICKS

❸ THE CALCULATOR MEMORIES

The calculator memories M0 to M9 are available as floating-point variables in OPL. You might use them to allow OPL access to results from the calculator, particularly if you use OPL procedures from within the calculator.

It's best not to use them as a substitute for declaring your own variables. Your OPL program might go wrong if another running OPL program uses them, or if you use them yourself in the calculator.

RUNNING A PROGRAM TWICE

Although you may never need to, you **can** run more than one copy of the same translated OPL module **at the same time**. There are two ways:

- Use 'Copy file' in the System screen to make a new copy of the module, with a different filename. Then run both files.

- **For the Series 3c only**, run the file as normal. Then move the highlight to under the RunOpl icon, press Tab to show the file selector, and pick the name of the translated module again.

❺ FOREGROUND AND BACKGROUND

- SETFLAGS $10000 tells the Series 5 to send a “machine switch on” event to the current program, whenever the Series 5 switches on, even if this program is in the background. If required, use it just once at the start of your program.
- The System OPX procedure SETFOREGROUND: brings the current program to the foreground.
- The System OPX procedure SETBACKGROUND: sends it to the background again.

Note that each of these should be followed by `gUPDATE` to ensure they take effect immediately.

Note that when a program runs in the background it can stop the “automatic turn off” feature from working. However, as soon as the program waits for a keypress or an event, with GET/GET$ or GETEVENT32, auto-turn off can occur.

Auto-turn off can also occur if the program does a PAUSE (of 2 or more 20ths of a second), but only if the program has used SETACTIVE from System OPX to mark the program as inactive.

See the ‘OPX.pdf’ document for more information about System OPX procedures.

❸ FOREGROUND AND BACKGROUND

- CALL ($6c8d) tells the Series 3c to send a “machine switch on” event to the current program, whenever the Series 3c switches on, even if this program is in the background. If required, use it just once at the start of your program.
- CALL($198d,0,0) brings the current program to the foreground.
- CALL($198d,100,0) sends it to the background again.

Each of these should be followed by `gUPDATE` to ensure they take effect immediately.

This example program comes to the foreground and beeps whenever you turn the Series 3c on. Be careful to enter the CALL and GETEVENT statements exactly as shown.

```
PROC beepon:
   LOCAL a%(6)
   PRINT “Hello”
   CALL($6c8d) :GUPDATE
   WHILE 1
   DO
      GETEVENT a%()
      IF a%(1)=$404 :STOP :ENDIF :REM closedown
   UNTIL a%(1)=$403 :REM machine ON
   CALL($198d,0,0) :gUPDATE
   BEEP 5,300 :PAUSE 10 :BEEP 5,500
   CALL($198d,100,0) :gUPDATE
   ENDWH
ENDP
```

Note that when a program runs in the background it can stop the “automatic turn off” feature from working. However, as soon as the program waits for a keypress or an event, with GET/GET$ or GETEVENT, auto-turn off can occur.

Auto-turn off can also occur if the program does a PAUSE (of 2 or more 20ths of a second), but only if the program has used `CALL($138b)` (“unmark as active”)

❺ **On the Series 5, procedures are automatically cached** and the cache commands are not available on the Series 5. Therefore, the following section is only applicable to the Series 3c and Siena.

CACHEING PROCEDURES ON THE SERIES 3C AND SIENA

Without procedure cacheing, procedures are loaded from file whenever they are called and discarded when they return - LOADM simply loads a map listing procedure names and their positions in the module file so that they can be loaded fairly efficiently. The cache handling commands provide a method for keeping the code for a procedure loaded after it returns - it remains loaded until a procedure called later requires the space in the cache. The strategy is then to remove the least recently used procedures, making it more likely that all the procedures called together in a loop, for example, remain in the cache together, thus speeding up procedure calling significantly.

Cache handling keywords allow you to:

- create a cache of a specified initial and maximum size using CACHE init%,max%. You can specify these up to 32,767 bytes.

✎ If you use hex, you can even exceed this figure, if you need to - e.g. `CACHE $9000,$9000`. However, you cannot exceed the 64k total memory limit which each Series 3c process has.

- prevent loading and removal of procedures from the cache so that a given set of procedures can be guaranteed to remain in the cache using CACHE OFF. Procedures already in the cache are still used when cacheing is off. The loading and removal of procedures can subsequently be resumed using CACHE ON.
- tidy the cache by removing procedures that are no longer in use (i.e. procedures that have returned) using CACHETIDY.
- for advanced use during program development, further keywords are provided for inspecting the contents of the cache at any time (see CACHEHDR and CACHEREC).

CACHE SIZE

Cacheing procedures is not a cure all. Care should be taken that the cache size is sufficient to load all procedures required for a fast loop otherwise, for example, a large procedure may cause all the small ones in a loop to be removed and equally, a small one may require the large one to be removed, so that the cache provides no benefit at all. In fact, the overhead needed for cache management can then make your program less efficient than having no cache at all. If the maximum cache size you can have is limited, careful use of `CACHE OFF` should prevent such problems at the expense of not fitting all the procedures in the loop in the cache. `CACHE OFF` is implemented very efficiently and calling it frequently in a loop should not cause much concern.

To guarantee that there is enough memory for a given cache size, create the cache passing that value as the initial size using `TRAP CACHE init%,max%`. TRAP ensures that if the cache creation succeeds, ERR returns zero and otherwise the negative ‘Out of memory’ error is raised. After creation, the cache will grow as required up to the maximum size `max%` or until there is not enough free memory to grow it. On failure to grow the cache, any procedures which will not fit into the existing cache, even when unused procedures are removed, are simply loaded without using the cache and are discarded when they return.

If you want to ensure a certain minimum cache size, say 10000 bytes, but do not care how large it grows, you could use `TRAP CACHE 10000,$ffff` so that the cache just grows up to the limits of memory. For a relatively small program, you might want to load the whole program into cache by making the cache size the same size as the module. This will in fact be a little larger than required, unnecessarily including a procedure name table and module file header which are not loaded into the cache. The minimum cache size is 2000 bytes, which is used if any lower value is specified. If the maximum size specified is less than the initial size, the maximum is set to the initial size. The maximum cache size cannot be changed once the cache has been created and an error is returned if you attempt to do so.

The initial cache size should ideally be large enough to hold all procedures that are to be cached simultaneously. There is no advantage in growing the cache from its initial size when you know that a certain minimum size is needed.

PROCEDURES IN UNLOADED MODULES

When a module is unloaded, all procedures in it that are no longer in use are removed from the cache. Any procedure that is still in use, is hidden in the cache by changing its first character to lower case; when it finally returns, a hidden procedure is removed in the normal manner to make room for loading a new procedure when the cache is full. Note that it is considered bad practice to unload a module containing procedures that are still running - e.g. for a procedure to unload its own module.

CACHE TIMINGS

Calling an empty procedure that simply returns is approximately 10 times faster with a cache. This figure was obtained by calling such a procedure 10000 times in a loop, both with cacheing off and on, and subtracting the time taken running an empty loop in each case.

Clearly that case is one of the best for showing off the advantages of cacheing, and there is no general formula for calculating the speed gain. The procedures that benefit most will be those that need most module file access relative to their size in order to load them into memory. The programmer cannot reasonably write code taking this into account, so no further details are provided here.

The case described above does not require any procedures to be removed from the cache to make room for new procedures when the cache is full, and removal of procedures requires a fair amount of processing by the cache manager. If many procedures in a time-critical section of your program are loaded into the cache and not used often before removal, the speed gain may be less than expected - a larger cache may be called for to prevent too many removals.

It should be noted however, that even with the worst case of procedures being loaded into the cache for use just once before removal, having a cache is often superior to having no cache. This is because the cache manager reads module file data (required for loading the procedures into memory) in one block rather than a few bytes at a time and it is the avoidance of excessive file access which provides the primary speed gains for cacheing.

COMPATIBILITY MODE MODULES

Procedures in modules translated for the Series 3 cannot be loaded into the cache. On encountering such a procedure, the cache manager simply loads it without using the cache and discards it when it returns. The reason for this is that a few extra bytes of data are stored in the Series 3c modules which are needed by the cache manager.

POTENTIAL PROBLEMS IN EXISTING PROGRAMS

It is possible that previously undiscovered bugs in existing OPL programs are brought to light simply by adding code to use the cache.

Without cacheing, the variables in a procedure are followed immediately by the code for the procedure. Writing beyond the variables (for example reading too many bytes into the final variable using such keywords as gPEEKLINE or KEYA) would have written over the code itself but would have gone unnoticed unless you happened to loop back to the corrupted code. With a cached procedure, the code no longer follows your variables, so the corruption occurs elsewhere in memory, resulting quite probably in the program crashing.

CONTROLLING PROCEDURE CACHEING

`TRAP CACHE initSize%,maxSize%` creates a cache of a specified initial number of bytes, which may grow up to the specified maximum. If the maximum is less than the initial size, the initial size becomes the maximum. If growing the cache fails, normal loading without the cache is used. The ‘In use’ error (-9) is raised if a cache has been created previously or the ‘Out of memory’ error (-10) on failure to create a cache of the specified initial size - use the TRAP command if required. Procedure code and other information needed for setting up variables are loaded into the cache when the procedure is called. If there is no space in the cache and enough space can be regained, the least recently used procedures are removed. Otherwise the procedure is loaded in the normal way without cacheing.

Once a cache has been created, `CACHE OFF` prevents further cacheing, although the cache is still searched when calling subsequent procedures. `CACHE ON` may then be used to re-enable cacheing. Note that `CACHE ON` or `CACHE OFF` are ignored if used before `CACHE initSize%,maxSize%`.

TIDYING THE CACHE

CACHETIDY removes any procedures from the cache that have returned to their callers. This might be called after performing a large, self-contained action in the program which required many procedures. Using CACHETIDY will then result in speedier searching for procedures called subsequently. More importantly, it will prevent the procedures being unloaded one at a time when the need arises - it is very efficient to remove a set of procedures that are contiguous in the cache as is likely to be the case in this situation.

Note that a procedure which has returned is automatically removed from the cache if you unload the module it is in, so CACHETIDY needn’t be used for such a procedure.

GETTING CACHE INDEX HEADER INFORMATION

The CACHEHDR command is provided for advanced use and is intended for use during program development only.

`CACHEHDR ADDR(hdr%())` reads the current cache index header into the array `hdr%()` which must have at least 11 integer elements. Note that any information returned is liable to change whenever a procedure is called, so you cannot save these values over a procedure call.

If no cache has yet been created, `hdr%(10)=0` and the other data read is meaningless. Otherwise, the data read is as follows:

hdr%(1)	current address of the cache itself
hdr%(2)	number of procedures currently cached
hdr%(3)	maximum size of the cache in bytes
hdr%(4)	current size of the cache in bytes
hdr%(5)	number of free bytes in the cache

hdr%(6) total number of bytes in cached procedures which are freeable (i.e. not running)

hdr%(7) offset from the start of the cache index to the first free index record

hdr%(8) offset from start of cache index to most recently used procedure’s record; zero if none

hdr%(9) offset from start of cache index to least recently used procedure’s record; zero if none

hdr%(10) address of the cache index, or zero if no cache created yet

hdr%(11) non-zero if cacheing is on, and zero if it is off

The cache manager maintains an index for the cache consisting of an index header containing overall information for the whole cache as well as one index record for each procedure cached. All offsets mentioned above give the number of bytes from the start of the index to the procedure record specified. The index records for cached procedures form a doubly linked list, with one list beginning with the most recently used procedure (MRU), with offset given by `hdr%(8)`, and the other with the least recently used procedure (LRU) with offset given by `hdr%(9)`. A further singly linked list gives the offsets to free index records. The linkage mechanism is described in the discussion of CACHEREC below.

GETTING A CACHE INDEX RECORD

The CACHEREC command is provided for advanced use and is intended for use during program development only.

`CACHEREC ADDR(rec%()),offset%` reads the cache index record (see the description of CACHEHDR above) at `offset%` into array `rec%()` which must have at least 18 integer elements. `offset%=0` specifies the most recently used (MRU) procedure’s record if any and `offset%<0` the least recently used procedure (LRU) procedure’s record if any.

The data returned by CACHEREC is meaningless if no cache exists (in which case `rec%(17)=0`) **or if there are no procedures cached yet** (when `hdr%(8)=0` as returned by CACHEHDR).

Each record gives the offset to both the more recently used and to the less recently used procedure’s record in the linked lists, except for the MRU and the LRU procedures’ records themselves which each terminate one of the lists with a zero offset. The first free index record (see CACHEHDR above) starts the free record list, in which each record gives the offset of the next free record or zero offset to terminate the list. To “walk” the cache index, you would always start by calling CACHEREC specifying either the MRU or LRU record offset, and use the values returned to read the less or more recently used procedure’s record respectively. Note that any information returned is liable to change whenever a procedure is called, so you cannot save these values over a procedure call.

For the free cell list, only `rec%(1)` is significant, giving the offset of the next free index record. For the records in the lists starting with either the LRU or MRU record, the data returned in `rec%()` is:

rec%(1) offset to less recently used procedure’s record or zero if on LRU

rec%(2) offset to more recently used procedure’s record or zero if on MRU

rec%(3) usage count zero if not running

rec%(4) offset in cache itself to descriptor for building the procedure frame

rec%(5) offset in cache itself to translated code for the procedure

rec%(6) offset in cache itself to the end of the translated code for the procedure

rec%(7) number of bytes used by the procedure in the cache itself

rec%(8-15) leading byte counted procedure name, followed by some private data rec%(16)   address of the procedure’s leading byte counted module name

rec%(17)   address of the cache index, or zero if no cache created yet

rec%(18)   non-zero if cacheing is on, and zero if it is off

For example, to print the names of procedures and their sizes from MRU to LRU:

```
CACHEHDR ADDR(hdr%())
IF hdr%(10)=0
   PRINT “No cache created yet”
   RETURN
ENDIF
IF hdr%(8)=0                          REM MRU zero?
   PRINT “None cached currently”
   RETURN
ENDIF
rec%(1)=0                             REM MRU first
DO
   CACHEREC ADDR(rec%()),rec%(1)      REM less recently used proc
   PRINT PEEK$(ADDR(rec%(8))),rec%(7) REM name and size
UNTIL rec%(1)=0
```

❺ **For Sprite handling on the Series 5, see the ‘OPX.pdf’ document.** Note, however, that the basic idea of sprite handling remains the same for the Series 5, and you may find some of the information given below helpful.

SPRITE HANDLING ON THE SERIES 3C AND SIENA

HOW SPRITES WORK

OPL includes a set of keywords for handling a *sprite* - a user-defined black/grey/white graphics object of variable size, displayed on the screen at a specified position.

The sprite can also be *animated* - you can specify up to 13 *bitmap-sets* which are automatically presented in a cycle, with the duration for each bitmap-set specified by you. Each bitmap-set may be displayed at a specifiable offset from the sprite’s notional position.

The 13 bitmap-sets are each composed of up to six bitmaps. The set pixels in each bitmap specify one of the following six actions: black pixels to be drawn; black pixels to be cleared; black pixels to be inverted; grey pixels to be drawn; grey pixels to be cleared; or grey pixels to be inverted. The bitmaps in a set must have the same size.

All the bitmaps in a set are drawn to the screen together and displayed for the specified duration, followed by the next set, and so on.

If you do not specify that a pixel is to be drawn, cleared or inverted, the background pixel is left unchanged.

Black pixels are drawn “on top of” grey pixels, so if you clear/invert just the grey pixels in the sprite they will be hidden under any pixels set black. So to clear/invert pixels on a background which has both grey and black pixels set, you need to clear/invert both black and grey pixels in the sprite.

✎ The pixels of one colour (black or grey) which are set in one bitmap of the bitmap-set should not overlap with those of the same colour which are set in another bitmap in the same bitmap-set. This is because the order in which the bitmaps are applied is undefined. So, for example, do not specify that pixel (0,0) should have the black pixel both drawn and cleared.

WHY USE SPRITES?

A sprite is useful for displaying something in foreground without having to worry about restoring the background display. A sprite can also have any shape, leaving the background display all around it intact, and it can even be hollow - only the pixels specified by you are drawn, cleared or inverted. Typically only one bitmap-set containing two black bitmaps would be used - one for setting and one for clearing pixels.

You would not often use the sprite features in their full generality. In fact, more than one bitmap-set is needed only for animation and it is also seldom necessary to use all the available bitmaps in a single bitmap-set.

CREATING A SPRITE

`sprId%=CREATESPRITE` creates a sprite and returns the sprite ID.

APPENDING A BITMAP-SET TO A SPRITE

```
APPENDSPRITE tenths%,bitmap$()
```

```
APPENDSPRITE tenths%,bitmap$(),dx%,dy%
```

append a single bitmap-set to a sprite. These may be called up to 13 times for each sprite. APPENDSPRITE may be called only before the sprite is drawn, otherwise it raises an error. `tenths%` gives the duration in tenths of seconds for the bitmap-set to be displayed before going on to the next bitmap-set in the sequence. It is ignored if there is only one bitmap-set.

bitmap$()	contains the names of the six bitmap files in the set:
bitmap$(1)	for setting black pixels
bitmap$(2)	for clearing black pixels
bitmap$(3)	for inverting black pixels
bitmap$(4)	for setting grey pixels
bitmap$(5)	for clearing grey pixels
bitmap$(6)	for inverting grey pixels

Use `“”` to specify no bitmap. If `“”` is used for all the bitmaps in the set, the sprite is left blank for the specified duration.

The array must have at least 6 elements.

All the bitmaps in a single bitmap-set must be the same size, otherwise an ‘Invalid arguments’ error is raised on attempting to draw the sprite. Bitmaps in different bitmap-sets may differ in size. `dx%` and `dy%` are the (x,y) offsets from the sprite position (see CREATESPRITE) to the top-left of the bitmap-set with positive for right and down. The default value of each is zero.

Sprites may use considerable amounts of memory. A sprite should generally be created, initialised and closed in the same procedure to prevent memory fragmentation. Care should also be taken in error handling to close a sprite that is no longer in use.

Creating or changing a sprite consisting of many bitmaps requires a lot of file access and should therefore be avoided if very fast sprite creation is required. Once the sprite has been drawn, no further file access is performed (even when it is animated) so the number of bitmaps is no longer important.

DRAWING A SPRITE

`DRAWSPRITE x%,y%` draws a sprite in the current window with top-left at pixel position `(x%,y%)`. The sprite must previously have been initialised using APPENDSPRITE or the ‘Resource not open’ error (-15) is raised. If any bitmap-set contains bitmaps with different sizes, DRAWSPRITE raises an ‘Invalid arguments’ error (-2).

CHANGING A BITMAP-SET IN A SPRITE

```
CHANGESPRITE index%,tenths%,var bitmap$()
```

```
CHANGESPRITE index%,tenths%,var bitmap$(),dx%,dy%
```

change the bitmap-set specified by `index%` (1 for the first bitmap-set) in the sprite using the supplied bitmap files, offsets and duration which are all used in the same way as for APPENDSPRITE.

CHANGESPRITE can be called only after DRAWSPRITE.

Note that if all or many bitmap-sets in the sprite need changing or if each bitmap-set consists of many bitmaps, the time required to read the bitmaps from file may be considerable, especially if fast animation is in progress. In such circumstances, you should think about closing the sprite and creating a new one, which will often be more efficient.

POSITIONING A SPRITE

`POSSPRITE x%,y%` sets the position of the sprite to `(x%,y%)`.

CLOSING A SPRITE

`CLOSESPRITE sprId%` closes the sprite with ID `sprId%`.

SPRITE EXAMPLE

The following code illustrates all the sprite handling keywords using a sprite consisting of just two bitmap-sets each containing a single bitmap.

```
PROC sprite:
   LOCAL bit$(6,6),sprId%
   crBits:                            REM create bitmap files
   gAT gWIDTH/2,0
   gFILL gWIDTH/2,gHEIGHT,0           REM fill half of screen
   sprId%=CREATESPRITE
   bit$(1)=““ :bit$(2)=““
   bit$(3)=“cross”                    REM black cross, pixels inverted
   bit$(4)=““ :bit$(5)=““ :bit$(6)=“
   APPENDSPRITE 5,bit$(),0,0          REM cross for half a second
   bit$(1)=“” :bit$(2)=““ :bit$(3)=“”
   bit$(4)=“” :bit$(5)=““ :bit$(6)=“”
   APPENDSPRITE 5,bit$(),0,0          REM blank for half a second
   DRAWSPRITE gWIDTH/2-5,gHEIGHT/2-5  REM animate the sprite
   BUSY “flash cross, c”,3            REM no offset (‘c’ for central)
   GET bit$(3)=“box”                  REM black box, pixels inverted
   CHANGESPRITE 2,5,bit$(),0,0        REM in 2nd bitmap-set
   BUSY “cross/box, c/c”,3            REM central/central
   GET
   CHANGESPRITE 2,5,bit$(),40,0       REM offset by 40 pixels right
```

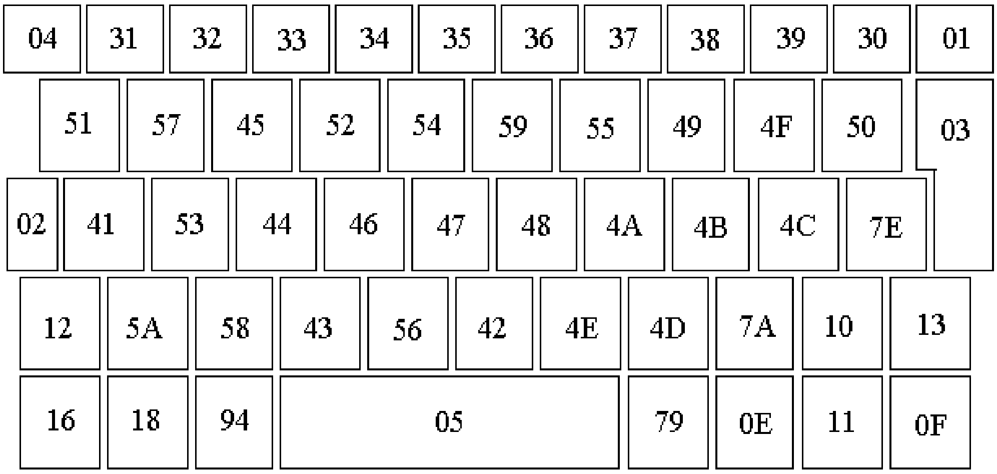
```
  BUSY “cross/box, c/40”,3           REM central/40
  GET
  bit$(3)=““                         REM Remove the cross in set 1
  CHANGESPRITE 1,3,bit$(),0,0        REM display for 3/10 seconds
  BUSY “flash box, 40”,3             REM box at offset 40 still
  GET
  bit$(3)=“cross”
  CHANGESPRITE 1,5,bit$(),0,0        REM cross centralised - set 1
  bit$(3)=“box”
  CHANGESPRITE 2,5,bit$(),0,0        REM box centralised - set 2
  BUSY “Escape quits”
  DO
    POSSPRITE RND*(gWIDTH-11),RND*(gHEIGHT-11)
    REM move sprite randomly
    PAUSE -20                        REM once a second
  UNTIL KEY = 27
  CLOSESPRITE sprId%
ENDP

PROC crBits:
  REM create bitmap files if they don’t exist
  IF NOT EXIST(“cross.pic”) OR NOT EXIST(“box.pic”)
    gCREATE(0,0,11,11,1,1)
    gAT 5,0 :gLINEBY 0,11
    gAT 0,5 :gLINEBY 11,0
    gSAVEBIT “cross”
    gCLS
    gAT 0,0
    gBOX gWIDTH,gHEIGHT
    gSAVEBIT “box”
    gCLOSE gIDENTITY
  ENDIF
ENDP
```

SCANNING THE KEYBOARD DIRECTLY

It is sometimes useful to know which keys are being pressed at a given moment and also when a key is released. For example, in a game, a certain key might start some action and releasing the key might stop it.

❺ `GETEVENT32 ev&()` and `GETEVENTA32 ev&()` return the scan code of any key pressed to the third element of the array `ev&()`. It is the programmers responsibility to track to key currently being pressed. The scan codes are given in hex in the following representation of the keyboard:



04 31 32 33 34 35 36 37 38 39 30 01
51 57 45 52 54 59 55 49 4F 50 03
02 41 53 44 46 47 48 4A 4B 4C 7E
12 5A 58 43 56 42 4E 4D 7A 10 13
16 18 94 05 79 0E 11 0F

❸ `CALL($288e,ADDR(scan%()))` returns with the array `scan%()`, which must have at least 10 elements, containing a bit set for keys currently being pressed.

Every key on the keyboard is represented by a unique bit. This includes the modifier keys (Shift, Control etc.) and the application buttons (System, Data, Word etc.).

A set bit simply signifies a pressed key - a key pressed on its own gives one bit set; that same key with a modifier gives the same bit set with another bit for the modifier; the modifier on its own gives the same modifier bit on its own.

The following table lists each key (according to the text printed on the physical key itself), the `scan%()` array element for that key and the hexadecimal bit mask to be ANDed with that array element to check whether the key is being pressed.

key	`scan%()`	*mask*	*key*	`scan%()`	*mask*
System	5	$200	Data	4	$200
Word	6	$200	Agenda	2	$200
Time	1	$200	World	3	$200
Calc	2	$100	Sheet	1	$100
Esc	8	$100	1	8	$02
2	8	$04	3	6	$40
4	5	$04	5	5	$08

6	8	$08	7	4	$40
8	3	$08	9	3	$10
0	2	$10	+	2	$08
Delete	3	$01	Tab	1	$04
Q	7	$02	W	7	$20
E	6	$20	R	5	$02
T	5	$10	Y	1	$08
U	4	$20	I	3	$04
O	3	$20	P	2	$20
-	2	$04	Enter	1	$01
Control	3	$80	A	7	$04
S	7	$10	D	6	$10
F	6	$02	G	5	$20
H	8	$40	J	4	$10
K	3	$02	L	3	$40
*	2	$40	/	2	$02
Left shift	2	$80	Z	7	$08
X	7	$40	C	6	$08
V	6	$04	B	5	$40
N	1	$40	M	4	$08
,	4	$02	.	8	$10
Up	8	$20	Right shift	4	$80
Psion	1	$80	Menu	6	$80
◆	5	$80	Space	5	$01
Help	4	$04	Left	1	$10
Down	1	$20	Right	1	$02

For example, pressing Tab sets bit 2 of `scan%(1)`, pressing Control sets bit 7 of `scan%(3)` and pressing both together sets both these bits. So Tab is being pressed if `scan%(1) AND $04` is non-zero, and Control is being pressed if `scan%(3) AND $80` is non-zero.

A possible strategy for scanning the keys might be to wait for any key of interest using GETEVENT or GET (allowing switch off and less intensive use of the battery), start the required action, which is to be continued only while the key is being pressed, scan the keyboard, as discussed above, until the key is released, and then stop the action, wait for the next key and repeat.

Note that the key returned by GETEVENT or GET is not precisely synchronised with those scanned, so once you have waited for a relevant key you should scan for all the keys pressed, ignoring the keycode returned by GETEVENT or GET.

I/O FUNCTIONS AND COMMANDS

✍ Note that one of the fundamental differences between the Series 3c and the Series 5 is that while any OPL program on the Series 3c has a 64K limit on the memory it may use, the memory which an OPL program may use on the Series 5 is unlimited up to the constraint placed by the machine itself. Therefore while 16-bit addresses are sufficient on the Series 3c, the Series 5 requires 32-bit addressing. As far as I/O functions are concerned, this means that any argument which is an address is an integer on the Series 3c, while it must be a long integer on the Series 5. See also the later section on ’32-bit addressing’.

OPL includes powerful facilities to handle input and output (‘I/O’). These functions and commands can be used to access all types of files on the Psion, as well as various other parts of the low-level software.

This section describes how to open, close, read and write to files, and how to set the position in a file. The data file handling commands and functions have been designed for use specifically with data files. **The I/O functions and commands are designed for general file access.** You don’t need to use them to handle data files.

These are powerful functions and commands and they must be used with care. Before using them you must read this document closely and have a good grounding in OPL in general.

ERROR HANDLING

You should have a good understanding of error handling before using the I/O functions.

The functions in this section never raise an OPL error message. Instead they return a value - if this is less than zero an error has occurred. It is the responsibility of the programmer to check all return values, and handle errors appropriately. Any error number returned will be one of those in the list given in the ‘Errors.pdf’ document. You can use ERR$ to display the error as usual.

HANDLES

Many of these functions use a *handle*, which must be a long integer variable on the Series 5 and an integer variable on the Series 3c (see the note above). IOOPEN assigns this handle variable a value, which subsequent I/O functions use to access that particular file. Each file you IOOPEN needs a **different** handle variable.

VAR VARIABLES

In this section, `var` denotes an argument which should normally be a LOCAL or GLOBAL variable. (Single elements of arrays may also be used, but not field variables or procedure parameters.) Where you see `var` the **address** of the variable is passed, not the value in it. (This happens **automatically**; don’t use ADDR yourself.)

In many cases the function you are calling passes information back by setting these `var` variables.

`var` is just to show you where you must use a suitable variable; you don’t actually type it.

❸ For example: `ret%=IOOPEN(var handle%,name$,mode%)` in the syntax description indicates that `IOOPEN(h%,“abc”,0)` is correct (provided `h%` is a simple variable), but `IOOPEN(100,“abc”,0)` is incorrect.

❺ For example: `ret%=IOOPEN(var handle&,name$,mode%)` in the syntax description indicates that `IOOPEN(h&,“abc”,0)` is correct (provided `h&` is a simple variable), but `IOOPEN(100,“abc”,0)` is incorrect.

It is possible, though, that you already have the address of the variable to use. It might be that this address is held in a field variable, or is even a constant value, but the most common situation is when the address was passed as a parameter to the current procedure.

If you add a # prefix to a `var` argument, this tells OPL that the expression following is the address to be used, not a variable whose address is to be taken.

Here is an example program:

```
PROC doopen:(phandle&, name$, mode%)
   LOCAL error%
   REM IOOPEN, handling errors
   error% = IOOPEN(#phandle&, name$, mode%)
   IF error% : RAISE error% : ENDIF
ENDP
```

The current value held in `phandle&` is passed to IOOPEN. You might call `doopen:` like this:

```
local filhand%,...
...
doopen:(addr(filhand%),“log.txt”,$23)
...
```

The `doopen:` procedure calls IOOPEN with the address of `filhand%`, and IOOPEN will write the handle into `filhand%`.

❸ Note that `phandle&` should be replaced by `phandle%` (i.e. integer rather than long integer) on the Series 3c.

If you ever need to add or subtract numbers from the address of a variable, use the UADD and USUB functions, or you run the risk of ‘Overflow’ errors.

OPENING A FILE WITH IOOPEN

```
ret%=IOOPEN(var handle%,name$,mode%
```

or

❺ `ret%=IOOPEN(var handle%,address&,mode%)`

❸ `ret%=IOOPEN(var handle%,address%,mode%)`

for unique file creation.

Creates or opens a file (or device) called `name$` and sets `handle%` to the handle to be used by the other I/O functions.

`mode%` specifies how the file is to be opened. It is formed by ORing together values which fall into the three following categories:

MODE CATEGORY 1 - OPEN MODE

One and only one of the following values must be chosen from this category.

$0000 Open an existing file (or device). The initial current position is set to the start of the file.

$0001 Create a file which must not already exist.

$0002 Replace a file (truncate it to zero length) or create it if it does not exist.

$0003 Open an existing file for appending. The initial current position is set to the end of the file. For text format files (see $0020 below) this is the only way to position to end of file.

$0004 Creates a file with a unique name. For this case, you must use the address of a string instead of name$. This string specifies only the path of the file to be created (any file name in the string is ignored). The string at address% is then set by IOOPEN to the unique file name generated (this will include the full path). The string must be large enough to take 130 characters (the maximum length file specification). For example:

s$="C:\home\"   REM C should be replaced with M on Series 3c

IOOPEN(handle%,ADDR(s$),mode%)

This mode is typically used for temporary files which will later be deleted or renamed.

MODE CATEGORY 2 - FILE FORMAT

One and only one of the following values must be chosen from this category. When creating a file, this value specifies the format of the new file. When opening an existing file, make sure you use the format with which it was created.

$0000 The file is treated as a byte stream of binary data with no restriction on the value of any byte and no structure imposed upon the data. Up to 16K can be read from or written to the file in a single operation.

$0020 The file is treated as a sequence of variable length records. The records are assumed to contain text terminated by any combination of the CR and LF ($0D, $0A) characters. The maximum record length is 256 bytes and Control-Z ($1A) marks the end of the file.

❺ On the Series 5, all of these values are declared as constants 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.

MODE CATEGORY 3 - ACCESS FLAGS

Any combination of the following values may be chosen from this category.

$0100 Update flag. Allows the file to be written to as well as read. If not set, the file is opened for reading only. You **must** use this flag when creating or replacing a file.

$0200 Choose this value if you want the file to be open for *random* access (not sequential access), using the IOSEEK function.

$0400 Specifies that the file is being opened for sharing for example, with other running programs. Use if you want to read, not write to the file. If the file is opened for writing ($0100 above), this flag is ignored, since sharing is then not feasible. If not specified, the file is locked and may only be used by this running program.

CLOSING A FILE WITH IOCLOSE

Files should be closed when no longer being accessed. This releases memory and other resources back to the system.

```
ret%=IOCLOSE(handle%)
```

Closes a file (or device) with the handle `handle%` as set by IOOPEN.

READING A FILE WITH IOREAD

❺ `ret%=IOREAD(handle%,address&,maxLen%)`

❸ `ret%=IOREAD(handle%,address%,maxLen%)`

Reads up to `maxLen%` bytes from a file with the handle `handle%` as set by IOOPEN. `address&` (or `address%` on the Series 3c) is the address of a buffer into which the data is read. This buffer must be large enough to hold a maximum of `maxLen%` bytes. The buffer could be an array or even a single integer as required. No more than 16K bytes can be read at a time.

The value returned to `ret%` is the actual number of bytes read or, if negative, is an error value.

TEXT FILES

If `maxLen%` exceeds the current record length, data only up to the end of the record is read into the buffer. No error is returned and the file position is set to the next record.

If a record is longer than `maxLen%`, the error value ‘Record too large’ (-43) is returned. In this case the data read is valid but is truncated to length `maxLen%`, and the file position is set to the next record.

A string array `buffer$(255)` could be used, but make sure that you pass the address `ADDR(buffer$)+1` (`UADD(ADDR(buffer$),1)` on the Series 3c) to IOREAD. This leaves the leading byte free. You can then POKEB the leading byte with the count (returned to `ret%`) so that the string conforms to normal string format. See the example program.

BINARY FILES

If you request more bytes than are left in the file, the number of bytes actually read (even zero) will be less than the number requested. So if `ret%<maxLen%`, end of file has been reached. No error is returned by IOREAD in this case, but the next IOREAD would return the error value ‘End of file’ (-36).

To read up to 16K bytes (8192 integers), you could declare an integer array `buffer%(8192)`.

WRITING TO A FILE

❺ `ret%=IOWRITE(handle%,address&,length%)`

❸ `ret%=IOWRITE(handle%,address%,length%)`

Writes `length%` bytes stored in a buffer at `address&` (`address%` on the Series 3c) to a file with the handle `handle%`.

When a file is opened as a binary file, the data written by IOWRITE overwrites data at the current position.

When a file is opened as a text file, IOWRITE writes a single record; the closing CR/LF is automatically added.

POSITIONING WITHIN A FILE

`ret%=IOSEEK(handle%,mode%,var offset&)`

Seeks to a position in a file that has been opened for random access (see IOOPEN above).

`mode%` specifies how the argument `offset&` is to be used. `offset&` may be positive to move forwards or negative to move backwards. The values you can use for `mode%` are:

1 Set position in a binary file to the absolute value specified in `offset&`, with 0 for the first byte in the file.

2 Set position in a binary file to `offset&` bytes from the end of the file

3 Set position in a binary file to `offset&` bytes relative to the current position.

6 Rewind a text file to the first record. `offset&` is not used, but you must still pass it as a argument, for compatibility with the other cases.

IOSEEK sets the variable `offset&` to the absolute position set.

EXAMPLE - DISPLAYING A PLAIN TEXT FILE

This program opens a plain text file, such as one created with the ‘Export as text…’ option in the ‘File’ menu of the Program editor on the Series 5, or the ‘Save as’ option in the Word Processor or Database applications on the Series 3c, and types it to the screen. Press Esc to quit and any other key to pause the typing to the screen.

```
PROC ioType:
  LOCAL ret%,fName$(128),txt$(255),address&
  LOCAL handle%,mode%,k%
  PRINT “Filename?”, :INPUT fName$ : CLS
  mode%=$0400 OR $0020            REM open=$0000, text=$0020, share=$0400
  ret%=IOOPEN(handle%,fName$,mode%)
  IF ret%<0
    showErr:(ret%)
    RETURN
  ENDIF
  address&=ADDR(txt$)
  WHILE 1
  k%=KEY
  IF k%                           REM if keypress
    IF k%=27                      REM Esc pressed
        RETURN                    REM otherwise wait for a key
    ELSEIF GET=27
        RETURN                    REM Esc pressed
    ENDIF
  ENDIF
  ret%=IOREAD(handle%,address&+1,255)
  IF ret%<0
    IF ret%<>-36                  REM NOT EOF
        showErr:(ret%)
    ENDIF
    BREAK
  ELSE
    POKEB address&,ret%           REM leading byte count
    PRINT txt$
  ENDIF
  ENDWH
  ret%=IOCLOSE(handle%)
  IF ret%
    showErr:(ret%)
  ENDIF
  PAUSE -100 :KEY
ENDP

PROC showErr:(val%)
  PRINT “Error”,val%,err$(val%)
  GET
ENDP
```

Note that this example may be used for any of the machines. However, any address variables may be changed from long integers to integers if you are using the Series 3c or Siena.

ASYNCHRONOUS REQUESTS AND SEMAPHORES

This section provides the general background necessary for understanding how EPOC I/O devices can be accessed by an OPL application program.

ASYNCHRONOUS REQUESTS

Many operating system services are implemented in two steps:

1. make the service request (sometimes referred to as the *queuing* of a request)

2. wait for the requested operation to complete

In most cases, as well as providing functions for each step, the system provides a function containing both the above steps. Such functions are called *synchronous* because they automatically synchronise the requesting process by waiting until the operation has completed. The internal function that makes the request without waiting for completion is called an *asynchronous* function.

Examples of asynchronous request functions are:

IOC (or IOA) for requests on an open I/O channel

KEYA for requests on the keyboard channel

❺ GETEVENTA32 for requests of events from the widow server

❺ PLAYSOUNDA: (in System OPX) for requests to play sound files

The synchronous versions of the above functions are IOW, GET, GETEVENT32 and PLAYSOUND: respectively.

Applications use asynchronous requests in situations like the following:

1. make request A

2. make request B

3. wait for either of the requested operations to complete

Processes wait for the completion of asynchronous requests by waiting on their *I/O semaphore* where each request is associated with a *status word*.

THE I/O SEMAPHORE

When a process is created, the system automatically creates an *I/O semaphore* on its behalf (a more accurately descriptive name would have been the *asynchronous request semaphore*). After making one or more asynchronous requests using IOC or IOA, a process calls IOWAIT to wait on the I/O semaphore for one of the requests to complete. A typical application spends most of its time waiting on its I/O semaphore. For example, an interactive application process that is waiting for user input is waiting on the I/O semaphore.

Semaphores are provided to synchronise cooperating processes (where, in this context, a process includes a hardware interrupt). In OPL semaphores are used for synchronising the completion of asynchronous requests.

The semaphores in both the EPOC16 and EPOC32 operating systems are counting semaphores, having a signed value that is incremented by calling IOSIGNAL and decremented by calling IOWAIT. A semaphore with a negative value implies that a process must wait for the completion of an event.

The process or the hardware interrupt handler that implements the requested operation sends a signal to the process (using a generalised version of IOSIGNAL directed to the required process) to indicate that the operation has completed. If one or more wait handlers have been installed (wait handlers are described below), they may process the signal and re-signal using IOSIGNAL. In some cases, it is convenient for the requestor to use IOSIGNAL to signal itself and subsequently to process that signal in a central call to IOWAIT.

STATUS WORDS

Although the arguments to asynchronous request functions vary, they all take a so-called *status word* argument, which subsequently contains the status of the requested operation. On the Series 3c the status word is always a 16-bit integer, like `stat%`. On the Series 5, the status word is also a 16-bit integer, except when calling an asynchronous OPX procedure that specifically requires a 32-bit integer status word, such as PLAYSOUNDA: in System OPX.

All asynchronous requests exhibit the following behaviour:

1. While the request is pending, a 16-bit status word contains the negative -46.

   ❺ A 32-bit status word however contains &80000001. Const.oph provides constant definitions for these values. For details of how to use this file see the ‘Calling Procedures’ section of the ‘Basics.pdf’ document and see Appendix E in the ‘Appends.pdf’ document for a listing of it.

2. When the operation has completed, a value other than -46 is written to the status word. This value is zero or positive to indicate success, or a negative error number to indicate failure. For 16-bit status words, an OPL error value is used. See the ‘Errors.pdf’ document for a list of these.

   ❺ For 32-bit status words on the Series 5, an EPOC32 error value is used. The EPOC32 error values are listed in Appendix G in the ‘Appends.pdf’ document.

3. The requesting process’s I/O semaphore is signalled (after the status word has been written).

Making a request while a previous request on the same status word is still pending will normally result in a *system panic* where the program is killed immediately, without being able to trap the error.

When there are multiple requests, each request is associated with a different status word. After returning from IOWAIT, the caller typically *polls* each status word until one is found that contains other than -46. That completion is then processed (which might include renewing the asynchronous request) and IOWAIT is called again to process the next completion.

CANCELLING AN ASYNCHRONOUS REQUEST

Most asynchronous request functions made using IOC (or IOA) can be cancelled. To cancel such requests use the IOCANCEL function.

For asynchronous requests made using a mechanism other than IOC (or IOA), a specific cancelling function must be used instead:

- Use KEYC to cancel a KEYA request.
- ❺ Use GETEVENTC to cancel a GETEVENTA32 request.
- ❺ Use STOPSOUND&: to cancel a PLAYSOUNDA: request.

The following general principles apply to all functions that cancel an asynchronous request:

- the cancel precipitates the completion of the operation (it does not stop the operation from completing).
- the cancel may or not be effective (that is, the operation may complete naturally before the cancel is processed).
- after a cancel, you must still process the completion of the asynchronous request (typically by immediately calling IOWAITSTAT or IOWAITSTAT32, described below, to “use up” the signal). An exception to this rule is that GETEVENTC and KEYC themselves wait for the cancellation signal, so IOWAITSTAT must not be used.

A 16-bit status word is set to -48 when a request has been effective.

❺ A 32-bit status word is set to -3 (EPOC32’s error code for ‘Cancel error’) when a request using a 32-bit status word is cancelled.

WAITING FOR A PARTICULAR COMPLETION

When waiting for the completion of a *particular* asynchronous request, the wait on the I/O semaphore must be sure that it is not fooled into a premature return by the completion of any other pending asynchronous request. This is done by calling IOWAITSTAT which behaves in a similar way to IOWAIT except that it only returns when the associated status word is other than -46.

❺ For a 32-bit status word request, IOWAITSTAT32 is used instead, again behaving in a similar way to IOWAIT except that it only returns when the associated 32-bit status word is other than &80000001.

In general, IOWAITSTAT (IOWAITSTAT32) is a safer option than IOWAIT to “use up” the signal resulting from the cancelled operation. If the cancel is not immediately effective and another completion causes IOWAIT to return, the program could continue and make another request before the cancelled operation completes (which would result in a process panic).

A FIRST EXAMPLE USING ASYNCHRONOUS I/O

In the following example, the opened asynchronous timer `timChan%` is used to construct a synchronous function which attempts to write the passed string to the opened serial channel `serChan%`. If it takes more that 5 seconds to complete the write, the procedure raises the ‘inactivity’ error -54. For simplicity it is assumed that there are no other outstanding events which could complete and that both requests started successfully. Thus it is certain that on return from the call to IOWAIT, one of the two asynchronous requests has completed.

```
PROC strToSer:(inStr$)
   LOCAL str$(255)            REM local copy of inStr$ needed for ADDR
   LOCAL len%,timStat%,serStat%,timeout%,ret%,pStr&
                              REM request asynchronous serial write
   str$=inStr$
   pStr&=ADDR(str$)+1         REM pointer to string skipping leading count
byte
   len%=LEN(str$)
   IOC(serChan%,2,serStat%,#pStr&,len%)
                              REM request asynchronous serial write
   timeout%=50                REM 5 second timeout
   IOC(timChan%,1,timStat%,timeout%)            REM Relative timer -
function 1
   IOWAIT
   IF serStat%=-46            REM must have timed out
      IOCANCEL(serChan%)      REM cancel serial request
      IOWAITSTAT serStat%     REM use up the signal
      RAISE -54               REM inactivity timeout
   ENDIF
   IOCANCEL(timChan%)         REM cancel timer request
   IOWAITSTAT timStat%        REM use up the signal
ENDP
```

3 WAIT HANDLERS

Wait handlers are functions that handle the completion of asynchronous requests from within IOWAIT (or IOWAITSTAT). Active wait handler functions are called just before IOWAIT would have otherwise returned.

Many I/O devices install a *device wait handler* when the device is opened.

Wait handlers are only called when the process calls IOWAIT (or a function such as IOWAITSTAT that calls IOWAIT). While an application performs a computationally intensive task that takes an extended time, it should consider calling IOYIELD (which effectively calls IOSIGNAL followed by IOWAIT) to allow any installed wait handlers to be called. Application programs must never assume that no wait handlers have been installed.

POLLING RATHER THAN WAITING

In a multi-tasking operating system it is extremely anti-social to wait for an operation to complete by polling the status word in a tight loop rather than call IOWAIT (because the polling will “hog” the processor to no benefit). However, when there is useful work to be done between each poll, it can be appropriate to poll - for example, to check periodically for user input while performing an extended calculation.

When a poll detects a completed status word, it is still obligatory to “use up” the signal by calling IOWAIT (otherwise you will get a “stray signal” later).

⑤ POLLING ON THE SERIES 5

The status word will be set only when either IOWAIT or IOYIELD is called. For a 32-bit status word, call IOWAITSTAT32 instead.

For example:

```
...
IOC(handle%,func%,stat%,#pBuf&,len%)
DO
     calc:    REM perform part of some calculation until request complete
     IOYIELD  REM allow status word to be set
UNTIL stat%<>-46
...
```

③ POLLING ON THE SERIES 3C AND SIENA

If this approach is used, you should be aware that in some cases asynchronous requests are completed by a wait handler and it is necessary to call IOYIELD *before* each poll to give any wait handlers a chance to run. For example:

```
...
IOC(handle%,func%,stat%,#pBuf%,len%)
DO
     calc:    REM perform part of some calculation until request complete
     IOYIELD  REM allow handler to run
UNTIL stat%<>-46
...
```

This case occurs when making requests on a device driver that services hardware interrupts. For example, the serial port driver services the hardware interrupt generated by the receipt of a serial frame. A hardware interrupt handler cannot write directly to the data segment of the requesting process because that data segment may be moving when the interrupt occurs. Instead, the interrupt handler must write first to a fixed memory location (either in the operating system variables if it is an in-built driver or in a device segment if it is an external driver) and then signal the I/O semaphore of the requesting process. When the requesting process next calls IOWAIT (directly or indirectly through say IOYIELD), the device driver's wait handler is called to copy the data safely to the process data segment.

Device drivers that are implemented by a server process (for example, the window server) do not require a wait handler to complete operations.

I/O DEVICE HANDLING AND THE ASYNCHRONOUS REQUEST FUNCTIONS

The following I/O functions provide access to devices. A full description is not within the scope of this User Guide, since these functions require **extensive** knowledge of the Series 5 or Series 3c operating systems and related programming techniques. The syntax and argument descriptions are provided here for completeness.

The previous section explains in detail the semantics of asynchronous I/O. In the descriptions of the asynchronous I/O functions below, a careful reading of that section is assumed.

`ret%=IOW(handle%,func%,var arg1,var arg2)`

The device driver opened with `handle%` (as returned by IOOPEN) performs the synchronous I/O function `func%` with the two further arguments. The size and structure of these two arguments is specified by the particular device driver's documentation.

`IOC(handle%,func%,var stat%,var a1,var a2)`

`IOC(handle%,func%,var stat%,var a1)`

`IOC(handle%,func%,var stat%)`

Make an I/O request with guaranteed completion. The device driver opened with `handle%` (as returned by IOOPEN) performs the asynchronous I/O function `func%` with up to two further arguments. The size and structure of these arguments is specified by the particular device driver's documentation.

As explained in detail in the previous section, *asynchronous* means that the IOC returns immediately, and the OPL program can carry on with other statements. `status%` will always be set to -46, which means that the function is still pending.

When, at some later time, the function completes, `status%` is automatically changed. (For this reason, `status%` should usually be global since if the program is still running, `status%` must be available when the request completes, or the program will probably crash). If `status%`>=0, the function completed without error. If `status%<0`, the function completed with error. The return values and error codes are specific to the device driver.

If an OPL program is ready to exit, it does not have to wait for any signals from pending IOC calls.

`ret%=IOA(handle%,func%,var status%,var arg1,var arg2)`

The device driver opened with `handle%` (as returned by IOOPEN) performs the asynchronous I/O function `func%` with two further arguments. The size and structure of these two arguments is specified by the particular device driver's documentation.

This has the same form as IOC, but the return value **cannot** be ignored. IOC is effectively the same as:

```
ret%=IOA(h%,f%,stat%,...)

IF ret%<0
   stat%=ret%
   IOSIGNAL
ENDIF
```

IOC allows you to assume that the request started successfully - any error is always given in the status word `stat%`. If there was an error, `stat%` contains the error code and the IOSIGNAL causes the next IOWAIT to return immediately as if the error occurred **after** completion. There is seldom a requirement to know whether an error occurred on starting a function, and IOC should therefore nearly always be used in preference to IOA.

`IOWAIT`

Wait for an asynchronous request (such as one requested by IOC, KEYA or GETEVENTA32) to complete. IOWAIT returns when **any** asynchronous I/O function completes. Check `status%` to find out which request has completed. Use IOWAITSTAT with the relevant status word to wait for a particular request to complete.

`IOSIGNAL`

Replace a signal of an I/O function's completion.

As shown in 'A simple example using asynchronous I/O' in the previous section, it is sometimes useful to construct a synchronous operation from two asynchronous operations by waiting for either to complete before returning. In that case it waited for either the serial write request or a timeout. As noted there, the example assumed that there were no other outstanding asynchronous requests. If there had been one or more asynchronous requests made before calling that procedure, such as a request to play a sound file, and if that request had completed before both the serial write and the timer request, the procedure would incorrectly assume that the timer had completed:

```
  IOWAIT
  IF serStat%=-46                REM must have timed out
    IOCANCEL(serChan%)           REM cancel serial request
    IOWAITSTAT serStat%          REM use up the signal
    RAISE -54                    REM inactivity timeout
  ENDIF
  IOCANCEL(timChan%)             REM cancel timer request
    IOWAITSTAT timStat%          REM use up the signal
```

To deal with this situation correctly, the procedure should instead check that one of the particular two requests it knows about has completed and re-signal to ‘put back’ the signal consumed for the sound file completion:

```
PROC strToSer:(inStr$)
  LOCAL str$(255)                REM local copy of inStr$ needed for ADDR
  LOCAL len%,timStat%,serStat%,timeout%,ret%,pStr&
  LOCAL signals%                 REM count of external signals
  LOCAL err%

  str$=inStr$
  pStr&=ADDR(str$)+1 REM ptr to string skipping leading count byte
  len%=LEN(str$)
  IOC(serChan%,2,serStat%,#pStr&,len%)
  REM request asynchronous serial write
  timeout%=50                    REM 5 second timeout
  IOC(timChan%,1,timStat%,timeout%)   REM relative timer - function 1
  WHILE 1                        REM forever loop
  IOWAIT                         REM wait for any completion
  IF timStat%<>-46               REM timed out
    IOCANCEL(serChan%)           REM cancel serial request
    IOWAITSTAT serStat%          REM use up the signal
    err%=-54                     REM inactivity timeout (raised
                                 REM below after re-signalling)
    BREAK                        REM stop waiting and re-signal
  ELSEIF serStat%<>-46           REM serial write complete
    IOCANCEL(timChan%)           REM cancel timer request
    IOWAITSTAT timStat%          REM use up the signal
    BREAK                        REM stop waiting and re-signal
  ELSE                           REM other unknown request
    signals%=signals%+1          REM count other signals
                                 REM loop again for next
  ENDIF
  ENDWH
  WHILE signals%>0               REM now re-signal any consumed
                                 REM external signals
  IOSIGNAL
  signals%=signals%-1
```

```
    ENDWH
    IF err%
        RAISE err%
    ENDIF
ENDP
```

IOSIGNAL is called only after exiting the IOWAIT loop, otherwise the signal would cause the IOWAIT to return immediately.

IOWAITSTAT var status%

Wait for a particular asynchronous function, called with IOC, to complete.

❺ `IOWAITSTAT32 var stat&`

Similar to IOWAITSTAT but takes a 32-bit status word. IOWAITSTAT32 should be called only when you need to wait for completion of a request made using a 32-bit status word when calling an asynchronous OPX procedure. `status&` will be &80000001 while the function is still pending, and on completion will be set to the appropriate EPOC32 error code, listed in Appendix G in the ‘Appends.pdf’ document. See also the ‘OPX.pdf’ document and the ‘Alphabetic listing’ section of the ‘Glossary.pdf’ document.

IOYIELD

Ensures that any asynchronous function is given a chance to run. Some devices are unable to perform an asynchronous request if an OPL program becomes computationally intensive, using no I/O (screen, keyboard etc.) at all. In such cases, the OPL program should use IOYIELD before checking its `status%` variable. This is **always** the case on the Series 5. IOYIELD is the equivalent of IOSIGNAL followed by IOWAIT - the IOWAIT returns immediately with the signal from IOSIGNAL, but the IOWAIT causes any asynchronous handlers to run.

IOCANCEL(handle%)

Cancels any outstanding asynchronous I/O request made using IOC (or IOA) on the specified channel, causing them to complete with the completion status word containing -48 (‘I/O cancelled’). The return value is always zero and may be ignored.

The IOCANCEL function is harmless if no request is outstanding (e.g. if the function completed just before cancellation requested).

GETEVENTA32(var status%,var event&())

This is an asynchronous window server read function. You must declare a long integer array with at least 16 elements. If a window server event occurs, the information is returned in `event&()` as described under GETEVENT32 in the alphabetic listing.

GETEVENTC(var status%)

Cancels a GETEVENTA32. Note that IOWAITSTAT should **not** be called after GETEVENTC - OPL consumes the GETEVENTC signal.

err%=KEYA(var status%,key%())

This is an asynchronous keyboard read function. You must declare an integer array with two elements here, `key%(1)` and `key%(2)` to receive the keypress information. If a key is pressed, the information is returned in this way:

- `key%(1)` is assigned the character code of the key.
- The least significant byte of `key%(2)` takes the key modifier, in the same way as KMOD 2 for Shift down, 4 for Control down and so on. KMOD cannot be used with KEYA.
- The most significant byte of key%(2) takes the count of keys pressed (0 or 1).

KEYA needs an IOWAIT in the same way as IOC.

SOME USEFUL IOW FUNCTIONS

IOW has this specification:

```
ret%=IOW(handle%,func%,var arg1,var arg2)
```

Here are some uses:

```
LOCAL a%(6)
```

```
IOW(-2,8,a%(),a%())          REM 2nd a% is ignored
```

senses the current text window (as set by the most recent SCREEN command) and the text cursor position **ignoring** any values already in the array `a%()`. This gives the same information as SCREENINFO, which should be used in preference (see the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document).

The first four elements are set to represent the offset of the current text window from the default text window. `a%(1)`is set to the `x`-offset of the top left corner of the current text window from the default text window’s top left corner and `a%(2)` to the `y`-offset of the top left corner of current text window. Similarly `a%(3)` and `a%(4)` give the offset of bottom right corner of text window from the bottom right corner of the default text window. For example, if the most recent SCREEN command was `SCREEN 10,11,4,5` the first four elements of the array `a%()` would be set to (3,4,13,15). The `x` and `y` positions of the cursor **relative to the current text window** are written to `a%(5)` and `a%(6)` respectively. All positions and offsets take `0,0`, not `1,1`, as the point at the top left.

```
LOCAL i%,a%(6)
i%=2
a%(1)=x1% :a%(2)=y1%
a%(3)=x2% :a%(4)=y2%
IOW(-2,7,i%,a%())
```

clears a rectangle at `x1%,y1%` (top left), `x2%,y2%` (bottom right). If `y2%` is one greater than `y1%`, this will clear part or all of a line.

EXAMPLE OF IOW SCREEN FUNCTIONS

The final two procedures in this module call the two IOW screen functions described beforehand. The rest of the module lets you select the function and values to use. It uses the technique used in the ‘Friendlier interaction’ section of the ‘GUI.pdf’ document of handling menus and short-cut keys by calling procedures with string expressions.

❸
```
PROC iotest:
   GLOBAL x1%,x2%,y1%,y2%
   LOCAL i%,h$(2),a$(5)
   x1%=2 :y1%=2
   x2%=25 :y2%=5                         REM our test screensize
   SCREEN x2%-x1%,y2%-y1%,x1%,y1%
   AT 1,1
   PRINT “Text window IO test”
   PRINT “Control-Q quits”               REM should be “Psion-Q” on 3c
   h$=“cr”                               REM our shortcut keys
   DO
      i%=GET
      IF i%=$122                         REM MENU key
            mINIT
            mCARD “Set”,”Rect”,%r
            mCARD “Sense”,”Cursor”,%c
            i%=MENU
         IF i% AND INTF(LOC(h$,CHR$(i%)))
            a$=“proc”+chr$(i%)
            @(a$):
         ENDIF
      ELSEIF i% AND $200                 REM shortcut key
         i%=(i%-$200)
         i%=LOC(h$,CHR$(i%))             REM One of ours?
         IF i%
            a$=“proc”+MID$(h$,i%,1)
            @(a$):
         ENDIF                           REM ignore other weird keypresses
      ELSE                               REM some other key, so return it
         PRINT CHR$(i%);
      ENDIF
   UNTIL 0
ENDP

PROC procc:
   LOCAL a&
   a&=iocurs&:
   PRINT “x”;1+(a& AND &ffff);
   PRINT “y”;1+(a&/&10000)
ENDP

PROC procr:
   LOCAL xx1%,yy1%,xx2%,yy2%
   LOCAL xx1&,yy1&,xx2&,yy2&
   dINIT “Clear rectangle”
   dLONG xx1&,“Top left x”,1,x2%-x1%
   dLONG yy1&,“Top left y”,1,y2%-y1%
   dLONG xx2&,“Bottom left x”,2,x2%-x1%
   dLONG yy2&,“Bottom left y”,2,y2%-y1%
   IF DIALOG
      xx1%=xx1&-1 :xx2%=xx2&-1
      yy1%=yy1&-1 :yy2%=yy2&-1
      iorect:(xx1%,yy1%,xx2%,yy2%)
   ENDIF
ENDP

PROC iocurs&:
   LOCAL a%(4),a&
   REM don’t change the order of these!
   a%(1)=x1% :a%(2)=y1%
   a%(3)=x2% :a%(4)=y2%
   IOW(-2,8,a%(),a%())             REM 2nd a% is ignored
   RETURN a&
ENDP

PROC iorect:(xx1%,yy1%,xx2%,yy2%)
   LOCAL i%,a%(6)
   i%=2 :REM “clear rect” option
   a%(1)=xx1% :a%(2)=yy1%
   a%(3)=xx2% :a%(4)=yy2%
   IOW(-2,7,i%,a%())
ENDP

❺ PROC iotest:
   GLOBAL x1%,x2%,y1%,y2%
   LOCAL i%,h$(2),a$(5)
   x1%=2 :y1%=2
   x2%=25 :y2%=5                    REM our test screensize
   SCREEN x2%-x1%,y2%-y1%,x1%,y1%
   AT 1,1
   PRINT “Text window IO test”
   PRINT “Control-Q quits”          REM should be “Psion-Q” on 3c
   h$=“cr”                          REM our shortcut keys
   DO
      i%=GET
      IF i%=$122                    REM MENU key
         mINIT
         mCARD “Set”,”Rect”,%r
         mCARD “Sense”,”Cursor”,%c
         i%=MENU
         IF i% AND INTF(LOC(h$,CHR$(i%)))
            a$=“proc”+chr$(i%)
            @(a$):
         ENDIF
         ELSEIF KMOD AND 4            REM Ctrl modification
            i%=i%+$40
            i%=LOC(h$,CHR$(i%))       REM One of ours?
            IF i%
               a$=“proc”+MID$(h$,i%,1)
               @(a$):
            ENDIF                     REM ignore other weird keypresses
         ELSE                         REM some other key, so return it
            PRINT CHR$(i%);
         ENDIF
      UNTIL 0
   ENDP

PROC procc:
   LOCAL a&
   a&=iocurs&:
      PRINT “x”;1+(a& AND &ffff);
      PRINT “y”;1+(a&/&10000)
   ENDP

   PROC procr:
      LOCAL xx1%,yy1%,xx2%,yy2%
      LOCAL xx1&,yy1&,xx2&,yy2&
      dINIT “Clear rectangle”
      dLONG xx1&,“Top left x”,1,x2%-x1%
      dLONG yy1&,“Top left y”,1,y2%-y1%
      dLONG xx2&,“Bottom left x”,2,x2%-x1%
      dLONG yy2&,“Bottom left y”,2,y2%-y1%
      IF DIALOG
         xx1%=xx1&-1 :xx2%=xx2&-1
         yy1%=yy1&-1 :yy2%=yy2&-1
         iorect:(xx1%,yy1%,xx2%,yy2%)
      ENDIF
   ENDP

   PROC iocurs&:
      LOCAL a%(4),a&
      REM don’t change the order of these!
      a%(1)=x1% :a%(2)=y1%
      a%(3)=x2% :a%(4)=y2%
      IOW(-2,8,a%(),a%())                  REM 2nd a% is ignored
      RETURN a&
   ENDP

   PROC iorect:(xx1%,yy1%,xx2%,yy2%)
      LOCAL i%,a%(6)
      i%=2 :REM “clear rect” option
      a%(1)=xx1% :a%(2)=yy1%
      a%(3)=xx2% :a%(4)=yy2%
      IOW(-2,7,i%,a%())
   ENDP
```

❺ The following two examples could not be used on the Series 5. The use of sound in OPL on the Series 5 is handled with the use of System OPX. See the ‘OPX.pdf’ document for further details of how to use this.

❸ ALARM EXAMPLE - IOC TO ALM:

The `ALM:` device provides access to alarms. When writing to it with IOW, IOC or IOA, you can use these two functions:

- function=1 - only the date (and no time) is shown on the screen when the alarm rings - e.g. Thu 5 Sep
- function=2 - the day and time are shown - e.g. Thu 11:54

In either case, you must pass these two arguments:

- An array of 2 long integers the first is the time for the alarm to go off, and the second is the time for which it is due. Both are given in seconds since midnight on 1/1/1970.
- A message, as a **zero-terminated** string of up to 64 characters.

This procedure asks for the information for an alarm, and sets it (as type 2 day and time to be shown when the alarm rings). If you press the Time button, and this is the next alarm to ring, it is shown as a RunOpl alarm.

```
PROC alm:
  LOCAL h%,a&(2),a$(64),b$(65),d&,t&,t2&,a%,r%,s%
  r%=IOOPEN(h%,”ALM:”,0)
  IF r%<0 :RAISE r% :ENDIF
  d&=DAYS(DAY,MONTH,YEAR)        REM today
  t&=DATETOSECS(1970,1,1,HOUR,MINUTE,0)
  dINIT “Set alarm”
  dTIME t&,”Time”,0,0,DATETOSECS(1970,1,1,23,59,59)
  dDATE d&,”Date”,d&,DAYS(31,12,2049)
  dTIME t2&,”Alarm advance time”,2,0,86399
  dEDIT a$,”Message”
  IF DIALOG
    a&(2)=86400*(d&-25567)+t&
    a&(1)=a&(2)-t2&
    b$=a$+CHR$(0)                REM zero-terminate the string
    IOC(h%,2,s%,a&(),#UADD(ADDR(b$),1))
  ENDIF
  IOCLOSE(h%)
ENDP
```

At the moment the alarm rings, either `s%` must still be available to take the status word set by this `ALM:` function, or the program must have exited. Otherwise the status word will be written to a random area of memory. So in this example, no error-checking is done after the IOC - the program just ends. So you would have to use this as a whole program itself - you must not call this procedure as written here from another procedure.

❸ DIALLING EXAMPLE - IOW TO SND:

The `SND:` device provides sound services on the Series 3c. One function, number 10, provides access to DTMF dialling. It requires these two arguments:

- The number to dial, as a zero-terminated string of up to 24 characters.
- An array of 2 integers. The first is the tone length (*256) plus the delay length, and the second is the pause length. All of these are specified in 1/32 of a second.

```
PROC dtmf:
  LOCAL h%,a$(24),b$(25),z%,r%,a%(2)
  r%=IOOPEN(h%,”SND:”,0)
  IF r%<0 :RAISE r% :ENDIF
  dINIT
  dEDIT a$,”Dial”
  IF DIALOG
    a%(1)=8+(256*8)
    a%(2)=48
    b$=a$+CHR$(0)
    r%=IOW(h%,10,#UADD(ADDR(b$),1),a%())
    IF r%<0 :RAISE r% :ENDIF
  ENDIF
  r%=IOCLOSE(h%)
  IF r%<0 :RAISE r% :ENDIF
ENDP
```

❺ **The following section does not apply to the Series 5.** The use of sound in OPL on the Series 5 is handled with the use of System OPX. See the ‘OPX.pdf’ document for further details of how to use this.

RECORDING AND PLAYING SOUNDS ON THE SERIES 3C AND SIENA

This section explains how to write a program which records sounds to a *sound file* using the in-built Series 3c microphone, and which plays these or pre-recorded sound files back.

SOUND FILE STRUCTURE

Series 3c sound files are files with a `.WVE` extension that contain a 32-byte header and a byte stream of digital sound which is sampled and played back at 8000 bytes per second (12-bit sound is converted to 8-bit using A-Law encoding).

The file header has the following format:

offset in file	*bytes*	*contents*
0	16	zero-terminated ‘ALawSoundFile**’
16	2	version of this format
18	4	number of 8-bit samples
22	2	trailing silence in ticks
24	2	repeats
26	6	spare bytes reserved for future use

The number of samples is the number of bytes following the header and should always be size of file less 32 for the header.

The silence in ticks is the number of system ticks of silence appended to each repeat on playback (in practice, you get at least 2 ticks between repeats). A system tick is 1/32 of a second or 250 samples.

The repeats are the number of times to repeat the sound on playback (0 and 1 are treated as the same).

You can truncate the sound file, change the number of repeats and the trailing silence by changing the file length and header using the I/O binary file handling functions (IOOPEN, IOW, IOSEEK, IOWRITE etc.) described elsewhere in this document.

For example, you can truncate the file to length `newLen&` using: `ret%=IOW(handle%,11,newLen&,#0)`

HOW TO RECORD AND PLAY SOUNDS

The following set of procedures perform asynchronous recording and playing of sounds.

```
PROC recorda:(pstat%,inname$,size%)
  LOCAL name$(128)
  name$=inname$+chr$(0)
  CALL($2186,UADD(ADDR(name$),1),size%,0,0,pstat%)
ENDP

PROC recordc:
  CALL($2386)
ENDP

PROC recordw%:(inname$,size%)
  LOCAL name$(128),p%,ret%
  p%=PEEKW($1c)+6            REM address of saved flags after CALL
  name$=inname$+chr$(0)
  ret%=CALL($2286,UADD(ADDR(name$),1),size%)
  IF PEEKW(p%) AND 1         REM carry set for error
    RETURN ret% OR $FF00     REM return error
  ENDIF
ENDP

PROC playa:(pstat%,inname$,ticks%,vol%)
  LOCAL name$(128)
  name$=inname$+chr$(0)
  CALL($1E86,UADD(ADDR(name$),1),ticks%,vol%,0,pstat%)
ENDP

PROC playc:
  CALL($2086)
ENDP

PROC playw%:(inname$,ticks%,vol%)
  LOCAL name$(128),p%,ret%
  p%=PEEKW($1c)+6            REM address of saved flags after CALL
  name$=inname$+chr$(0)
  ret%=CALL($1F86,UADD(ADDR(name$),1),ticks%,vol%)
  IF PEEKW(p%) AND 1        REM carry set for error
    RETURN ret% OR $FF00    REM return error
  ENDIF
ENDP
```

`recorda:(pstat%,inname$,size%)` and `recordw%:(inname$,size%)` respectively perform asynchronous and synchronous recording to file `inname$`. Any existing file is replaced. You can only record to the Internal disk or to a RAM SSD - you cannot record to a Flash SSD. (You can playback from a Flash SSD, however.)

`size%` specifies the maximum number of bytes to be recorded in units of 2048 bytes. To record for one second `size%=4`. This figure excludes the 32-byte header. Before recording, a file of length `32+size%*2048` bytes is created and there must actually be room on the disk for a file of that length.

`pstat%` is the address of the status word to take the completion code for asynchronous recording.

`recordc:` cancels recording and truncates the file to the actual length recorded before cancellation.

`playa:(pstat%,inname$,ticks%,vol%)` and `playw%:(inname$,ticks%,vol%)` respectively perform asynchronous and synchronous playing of `inname$`.

`inname$` should either be the filename of the sound file to play or a ‘*’ followed by just the name component of the sound file. If it is preceded by a ‘*’, the extension `.WVE` is assumed and the service automatically searches `ROM::` and the `\WVE` directories of `M:` (Internal disk), `A:` and `B:` (in that order). The `ROM:: .WVE` files have names `SYS$AL01`, `SYS$AL02` and `SYS$AL03`.

`ticks%` is the duration that the sound file will play back in system ticks. If it is shorter than the given sound then playback is truncated to that time. If `ticks%` is negative, in addition to truncating the playback of longer files, it pads out as necessary to that duration with silence. If duration is zero, it plays the file without truncation or padding. (Alarms use a parameter of -480 to truncate or pad out to 15 seconds.)

`volume%` is a number between 0 and 5 inclusive, with 0 being the loudest. On the Series 3c there are only 4 actual levels: 0/1, 2, 3, and 4/5. `pstat%` is the address of the status word to take the completion code for asynchronous playback. Playback will append periods of silence and repeat the sound as specified in the file header.

`playc:` cancels playing back a sound.

ALARMS

The system dialogs that are used to set alarms detect the presence of any `.WVE` files in the `\WVE` directory of any local directory (in practice, `A:`, `B:` and Internal) and make these available (by file name) as the sound of the alarm.

When an alarm with a `.WVE` file rings, `.WVE` file playback (including any repeats) is clipped to 15 seconds. If the `.WVE` file plays for less than 15 seconds (including any repeats), the 15 seconds is padded out with silence.

EXAMPLE OF RECORDING

The following asynchronously records `time%` seconds of sound to file `file$` or cancels the recording when any key is pressed. The section I/O device handling and asynchronous requests also in this document discusses the principles involved. The `recorda:` and `recordc:` procedures, from above, are used.

```
PROC record:(file$,time%)
  LOCAL sstat%,kstat%,key%(4),size%,ret%,signals%
  size%=time%*4
  recorda:(ADDR(sstat%),file$,size%)      REM async record
  IOC(-2,1,kstat%,key%())                 REM async key read
  WHILE 1
    IOWAIT                 REM wait for recording to complete, or a key
    IF sstat%<>-46                        REM if sound no longer pending
        IOCANCEL(-2)                      REM cancel key read
        IOWAITSTAT kstat%                 REM wait for cancellation
        IF sstat%<0
            gIPRINT “Error recording:”+err$(sstat%)
        ENDIF
        BREAK
    ELSEIF kstat%<>-46                    REM else if key pressed
        recordc:                          REM cancel record
        IOWAITSTAT sstat%                 REM wait for cancellation
        gIPRINT “Cancelled”
        BREAK
    ELSE               REM some async request made outside this PROC
        signals%=signals%+1               REM save it for later
    ENDIF
  ENDWH
  WHILE signals%
    IOSIGNAL                              REM put back foreign signals
    signals%=signals%-1
  ENDWH
ENDP
```

❸ SERIES 3C AND SIENA OPL DATABASE INFORMATION

`ODBINFO var info%()` is provided for advanced use only and allows you to use OS and CALL to call *DbfManager* interrupt functions not accessible with other OPL keywords.

The description given here will be meaningful only to those who have access to full SDK documentation of the DbfManager services, which explains any new terms. Since that documentation is essential for use of ODBINFO, no attempt is made here to explain these terms.

ODBINFO returns `info%()`, which must have four elements containing pointers to four blocks of data; the first corresponds to the file with logical name `A`, the second to `B` and so on.

Take extreme care not to corrupt these blocks of memory, as they are the actual data structures used by the OPL runtime interpreter.

A data block which has no open file using it has zero in the first two bytes. Otherwise, the block of data for each file has the following structure, giving the offset to each component from the start of the block and with offset 0 for the 1st byte of the block:

Offset	*Bytes*	*Description*
0	2	DBF system’s file control block (handle) or zero if file not open
2	2	offset in the record buffer to the current record
4	2	pointer to the field name buffer
6	2	number of fields
8	2	pointer to start of record buffer
10	2	length of a NULL record
12	1	non-zero if all fields are text
13	1	non-zero for read-only file
14	1	non-zero if record has been copied down
15	1	number of text fields
16	2	pointer to device name

EXAMPLE

To copy the *Descriptive Record* of logical file B to logical file C:

```
PROC dbfDesc:
  LOCAL ax%,bx%,cx%,dx%,si%,di%
  LOCAL info%(4),len%,psrc%,pdest%
  ODBINFO info%()
  bx%=PEEKW(info%(2))                 REM handle of logical file B
  ax%=$1700                           REM DbfDescRecordRead
  IF OS($d8,ADDR(ax%)) and 1
    RETURN ax% OR $ff00               REM return the error
  ENDIF
  REM the descriptive record has length ax%
  REM and is at address peekW(uadd(info%(2),8))
  IF ax%=0
    RETURN 0                          REM no DescRecord
  ENDIF
  len%=ax%+2                REM length of the descriptive record read
                                      REM + 2-byte header
  psrc%=PEEKW(uadd(info%(2),8))
  pdest%=PEEKW(uadd(info%(3),8))
  CALL($a1,0,len%,0,psrc%,pdest%)     REM copy to C’s buffer
  cx%=len%
  bx%=PEEKW(info%(3))                 REM handle of logical file C
  ax%=$1800                           REM DbfDescRecordWrite
  IF OS($d8,ADDR(ax%)) and 1
    RETURN ax% OR $ff00
  ENDIF
  RETURN 0                            REM success
ENDP
```

❺ **The following section does not apply to the Series 5.** OPXs provide a different mechanism for calling language extensions and creating object instances. It uses language extensions provided in separate EPOC32 DLLs written especially for OPL support. These DLLs can be added to the language by anyone at any time and have the file extension `.OPX`. Unlike procedures written in OPL, these procedures are as fast to call as built-in keywords. OPXs enable OPL programs to perform virtually any operation in EPOC32 which is available to a C++ program. OPX procedures are called in a similar way to user-defined OPL procedures.

SERIES 3C AND SIENA DYL HANDLING

This section contains a complete reference description of OPL's support for accessing previously created dynamic libraries (*DYLs*). These libraries have an object-oriented programming (*OOP*) user-interface and several have been built into the Series 3c ROM for use by the ROM applications. DYLs cannot be created using OPL.

Since a vast amount of documentation would need to be provided to describe the essential concepts of OOP and the services available in existing DYLs, no attempt is made to supply it here. This section simply introduces the syntax for all the OOP keywords supported in OPL with a brief description of each. Also, OOP terminology is used here without explanation, to cater for those who have previous experience of DYL handling in the 'C' programming language.

VAR ARGUMENTS

The use of `var` and # for arguments was discussed earlier in this document in the section 'I/O functions and commands'. The DYL handling keywords use `var` and # in the same way, for example:

```
ret%=SEND(pobj%,method%,var p1,var p2,var p3)
```

This is because many DYL methods need the address of a variable or of a structure to be passed to them.

When you use a LOCAL or GLOBAL variable as the `var` argument, the address of the variable is used. (You cannot use procedure parameters or field variables, for this reason.) **If you use a # before a `var` argument, though, the argument/value is used directly, instead of its address being used.**

If, for example, you need to call a method with `p1` the **address** of a long variable `a&`, `p2` the integer constant 3, and `p3` the address of a zero terminated string `"X"`, you could call it as follows:

```
s$="X"+CHR$(0)               REM zero terminate
p%=UADD(ADDR(s$),1)          REM skip leading count byte
ret%=SEND(pobj%,method%,a&,#3,#p%)
```

The address of `a&` is passed because there is no #. `3` and the value in `p%` are passed directly (no address is taken) because they are preceded by #.

LOADING A DYL

`ret%=LOADLIB(var cathand%,name$,link%)` loads and optionally links a DYL that is not in the ROM. If successful, writes the category handle to `cathand%` and returns zero. You would normally only set `link%` to zero if the DYL uses another DYL which you have yet to load in which case LINKLIB would subsequently be used. The DYL is shared in memory if already loaded by another process.

UNLOADING A DYL

`ret%=UNLOADLIB(cathand%)` unloads a DYL from memory. Returns zero if successful.

LINKING A DYL

`LINKLIB cathand%` links any libraries that have been loaded using LOADLIB. LINKLIB is not likely to be used much in OPL - pass `link%` with a non-zero value to LOADLIB instead.

FINDING A CATEGORY HANDLE GIVEN ITS NAME

`ret%=FINDLIB(var cathand%,name$)` finds DYL category `name$` (including `.DYL` extension) in the ROM. On success returns zero and writes the category handle to `cathand%`. To get the handle of a RAM-based DYL, use LOADLIB which guarantees that the DYL remains loaded in RAM. FINDLIB will get the handle of a RAM-based DYL but does not keep it in RAM.

CONVERTING A CATEGORY NUMBER TO A HANDLE

`cathand%=GETLIBH(catnum%)` converts a category number `catnum%` to a handle. If `catnum%` is zero, this gets the handle for OPL.DYL.

CREATING AN OBJECT BY CATEGORY NUMBER

`pobj%=NEWOBJ(catnum%,clnum%)` creates a new object by category number `catnum%` belonging to the class `clnum%`, returning the object handle on success or zero if out of memory. This keyword simply converts the category number supplied to a category handle using GETLIBH and then calls NEWOBJH.

CREATING AN OBJECT BY CATEGORY HANDLE

`pobj%=NEWOBJH(cathand%,clnum%)` creates a new object by category handle `cathand%` belonging to the class `clnum%`, returning the object handle on success or zero if out of memory.

SENDING A MESSAGE TO AN OBJECT

```
ret%=SEND(pobj%,method%)

ret%=SEND(pobj%,method%,var p1)

ret%=SEND(pobj%,method%,var p1,var p2)”

ret%=SEND(pobj%,method%,var p1,var p2,var p3)
```

send a message to the object `pobj%` to call the method number `method%`, passing between zero and three arguments depending on the requirements of the method, and returning the value returned by the selected method.

PROTECTED MESSAGE SENDING

```
ret%=ENTERSEND(pobj%,method%)

ret%=ENTERSEND(pobj%,method%,var p1)

ret%=ENTERSEND(pobj%,method%,var p1,var p2)

ret%=ENTERSEND(pobj%,method%,var p1,var p2,var p3)
```

send a message to an object with protection.

Methods which return errors by *leaving* must be called with protection.

ENTERSEND is the same as SEND except that, if the method leaves, the error code is returned to the caller; otherwise the value returned is as returned by the method.

Use ENTERSEND0 (described next) for methods which leave but do not return a value explicitly on success.

PROTECTED MESSAGE SENDING (RETURNS ZERO ON SUCCESS)

```
ret%=ENTERSEND0(pobj%,method%)
ret%=ENTERSEND0(pobj%,method%,var p1)
ret%=ENTERSEND0(pobj%,method%,var p1,var p2)
ret%=ENTERSEND0(pobj%,method%,var p1,var p2,var p3)
```

send a message to an object with protection and guarantee that the known value zero is returned on success. Otherwise ENTERSEND0 is the same as ENTERSEND.

Methods which return errors by *leaving* but return nothing (or *NULL*) on success must use ENTERSEND0. Besides providing protection, ENTERSEND0 also returns zero if the method did not leave, or the negative error code if it did.

If ENTERSEND were incorrectly used instead and the method completed successfully (i.e. without leaving), the return value would be random and could therefore be in the range of the error codes implying that the method failed.

DYNAMIC MEMORY ALLOCATION

For each running OPL program (or *process*) the operating system automatically allocates memory. On the Series 3c, this can grow up to a maximum of 32 bytes less than 64K. On the Series 5, there are no built-in memory limits and a module or application can use as much of the available memory as it requires. A corollary of this is that although the use of integers were sufficient for the storage of addresses on the Series 3c, this is no longer the case on the Series 5. Hence long integers must be used instead. See the ‘32-bit addressing’ section below.

MAXIMUM DATA SIZE IN A PROCEDURE

Although there is no built-in limit to the total amount of data in an OPL program on the Series 5, it is still not possible to declare variables in a procedure that use in total more than 65516 bytes. This allows the largest integer array in a procedure to have 32757 elements. This number decreases for any other variables, externals or for procedures called from the given procedure, as they all consume some of the procedure’s data space.

The maximum for the Series 3c is about 32758 bytes of data in total per procedure.

SERIES 5 32-BIT ADDRESSING

As described above, memory addresses outside the 64K address space need to be stored in long integers. Thus all keywords which support addresses take and return long integers on the Series 5. Note also that where # is used to specify the address of a variable that is listed in the syntax using `var`, the address is a long integer, so you would use `#address&`. See the ‘I/O functions and commands’ section above.

PORTING SERIES 3C PROGRAMS TO THE SERIES 5

To facilitate porting of OPL programs written on the Series 3c (and other earlier machines) to the Series 5, it is in fact possible to set a flag to emulate the Series 3c memory model using `SETFLAGS 1`. This will cause an 'Out of memory' error if an attempt is made to obtain an address beyond the 64K limit.

If this flag is set, the Series 5 checks that the 64K limit is not exceeded when:

- the variables for a procedure are allocated, on calling any procedure. If the address of any variable in the procedure would require more than 16 bits, the procedure will fail to be loaded and an 'Out of memory' error will be raised. This is evidently preferable to having, for example, p%=ADDR(i%) giving an 'Overflow' error, once the procedure has been loaded. The ideal solution is, of course, to port the program to use 32-bit addresses instead, but setting the flag is a quick solution for many applications that are known to require less than 64K.

- the value returned by the heap allocation keywords requires more than 16 bits.

Some existing Series 3c programs may at times attempt to exceed the 64K limit and deal with the 'Out of memory' error. To port such programs there are two choices. Either you can

- change all integers that contain addresses to be 32-bit

or

- set the flag that enforces the 64K limit using SETFLAGS.

Note that the use of the word "address" is in fact imprecise for the Series 5. Series 5 addresses are in fact offsets into OPL's variable-heap, so an address of 0 really means 0 offset relative to the base of OPL's heap. This allows OPL to perform the appropriate 64K limit tests. This method of addressing variables is referred to as "base-relative addressing".

Another potential problem may well concern you at this point. When, for example, displaying an OPL address in some diagnostic debugging code, if a variable `i%` or a heap cell has a base-relative offset of between 32768 and 65535, then `p%=ADDR(i%)` would produce an overflow error unless special measures were taken. The reason for this is that 16-bit integers are *signed* in OPL, so the range is -32768 to 32767. Hence, although 32768 fits in an *unsigned* 16-bit integer (hex $8000) it doesn't fit into a *signed* integer. OPL gets around this problem by *sign-extending* the result of ADDR and the heap functions when 64K restriction is in force, as follows:

- if the value returned is 32768 to 65535 (hex $8000 to $ffff), OPL treats the value as signed. So $8000 becomes &ffff8000 (or -32768), and $ffff becomes &ffffffff (or -1). These values can then be assigned to the signed p%.

- if the value is greater than or equal to 65536 (hex &10000) then it is not sign-extended and so assigning it to p% raises an 'Overflow' error as required.

As mentioned above, this conversion should always be totally transparent to your program. This is possible because the keywords that use these sign-extended addresses, all convert the value back to an unsigned 16-bit value before use.

There is one case, however, where porting is required even with the flag set for a 64K limit. This is when the long integer value returned by ADDR or from a heap-allocating function is passed directly into a user-defined procedure. An OPL procedure from the Series 3c taking an address parameter will probably have been written to take a 16-bit address. As the translator cannot necessarily know the type taken by a user-defined procedure, it cannot coerce the type returned by these functions to match that taken by the user-defined procedure. This means that a type violation error can be caused.

Example:

If a user-defined procedure, `userProc:(p%)` is called as follows:

```
userProc:(ADDR(i%))
```

the 32-bit integer returned by ADDR will cause a type violation, as it will not be coerced to a 16-bit integer. The solution is either to define `userProc:` to take a 32-bit integer, or to declare the prototype of `userProc:` as `userProc:(ADDR%)`. Either of these would allow translator coercion. See the ‘Calling Procedures’ section of the ‘Basics.pdf’ document and also EXTERNAL in the ‘Alphabetic Listing’ for more information on procedure prototyping in OPL on the Series 5.

PORTING SERIES 3C PROGRAMS TO THE SERIES 5 WITHOUT ENFORCING THE 64K LIMIT

If you are porting Series 3c OPL code (or code from earlier machines) to the Series 5 without using a SETFLAGS statement, it should be obvious that there are two points which you need to remember:

- any address variables which were previously specified as integers must be changed to long integers.
- as mentioned in the ‘Safe pointer arithmetic’ section above, UADD and USUB should **not** be used when you are using 32-bit addressing without the flag set to restrict to 64K. Use of these functions should be replaced with ordinary long integer arithmetic.

OVERVIEW OF MEMORY USAGE

The actual memory used by an OPL program depends on the requirements of the process and is automatically grown or shrunk as necessary.

❺ An OPL process contains several separate memory areas, but the only one of significant interest to the OPL programmer is the *OPL heap*, used to contain the OPL variables and dynamically allocated memory cells. The heaps of different processes are entirely separate - you need only concern yourself with the heap used in your own process.

❸ This memory is called the *process data segment* and contains all the data used by the process as well as some fixed length data at low memory in the segment needed by the operating system to manage the process and for other system data.

Although the data segment for an OPL process contains several components, the only component of significant interest to the OPL programmer is the *process heap*. This section describes several keywords for accessing the heap.

The heap is essentially a block of memory at the highest addresses in a process data segment, so that the operating system can grow and shrink the heap simply by growing and shrinking the data segment and without having to move other blocks of memory at higher addresses in the data segment. The heaps of different processes are totally independent - you need concern yourself only with the heap used in your own data segment.

THE HEAP ALLOCATOR

The heap allocator keywords are used to allocate, resize and free variable length memory cells from the OPL heap (the process heap on the Series 3c). Cells typically range in size from tens of bytes to a few kilobytes. Allocated cells are referenced directly by their address; they do not move to compact free space left by freed cells.

Heap allocator keywords are:

- ALLOC allocates a cell of specified size, returning its address.
- FREEALLOC frees a previously allocated cell, which is returned to the heap.
- REALLOC changes the size of a cell, returning its new address.
- ADJUSTALLOC opens or closes a gap in the middle of a cell (useful for insertion or deletion of cell content), changing the size of the cell as appropriate.
- LENALLOC returns the size of a cell.

THE HEAP STRUCTURE

Initially, the heap consists of a single free cell. After a number of calls to allocate and free cells, the heap typically consists of ranges of adjacent allocated cells separated by single free cells (which are linked). If a cell being freed is next to another free cell the two cells are automatically joined to make a single cell to prevent the free cell linked list from growing unnecessarily.

Writing beyond the end of a cell will corrupt the heap’s integrity. Such errors are difficult to debug because there is no immediate effect - the corruption is a “time bomb”. It will eventually be detected, resulting in the process exiting prematurely by a subsequent allocator call such as FREEALLOC.

GROWING AND SHRINKING THE HEAP

The heap is not fixed in size. The operating system can grow the heap to satisfy allocation requests or shrink it to release memory back to the system.

Allocation of cells is based on “walking” the free space list to find the first free cell that is big enough to satisfy the request. If no free cell is big enough, the operating system will attempt to grow the data segment to add more free space at the end of the heap.

If there is no memory in the system to accommodate growth or, on the Series 3c and Siena, if the data segment has reached its maximum size of (approximately) 64K, the allocate request fails. There are few circumstances when an allocate request can be assumed to succeed and calls to ALLOC, REALLOC and ADJUSTALLOC should have error recovery code to handle a failure to allocate.

LOST CELLS

There are cases in which programs allocate a sequence of cells which must either exist as a whole or not at all. If during the allocate sequence one of the later allocations fails, the previously allocated cells must be freed. If this is not done, the heap will contain unreferenced cells that consume memory to no purpose.

When designing multi-cell sequences of this kind, you should be mindful of the recovery code that must be written to free partially built multi-cell structures. The fewer the cells in such a structure, the simpler the recovery code is.

INTERNAL FRAGMENTATION

The free space in the heap is normally fragmented to some extent; the largest cell that can be allocated is substantially smaller than the total free space. Excessive fragmentation, where the free space is distributed over a large number of cells - and where, by implication, many of the free cells are small - should be avoided because it results in inefficient use of memory and reduces the speed with which cells are allocated and freed.

Practical design hints for limiting internal fragmentation are:

- Avoid using the heap for small, highly transient data structures for which ordinary variables are adequate. High frequency cycling through allocate and free pairs, “churns” the heap and leads to a long free space list.
- When you have a large number of variable length data structures - particularly when they are frequently resized, “granularise” them (i.e. round the allocation up to a multiple of some reasonable value) so that you decrease the chance of leaving small, unusable free space cells.

Although a small number of gaps are not too serious and should eventually disappear in most cases anyway, the new heap allocating keywords provide ample opportunity to fragment the heap. Provided that you create and free cells in a careful and structured way, where any task needing the allocator frees them tidily on completion, there should not be a problem.

THE OPL RUNTIME INTERPRETER AND THE HEAP

❸ The OPL runtime interpreter, which actually runs your program, uses the same data segment and heap as your program and makes extensive use of the heap. It is very important that you should understand the interpreter’s use of the heap - at least to a limited extent to avoid substantial internal fragmentation as described above.

Whenever an OPL procedure is called, a cell is allocated to store data required by the interpreter to manage the procedure. The same cell contains all the variables that you have declared in the procedure. On the Series 3c, when cacheing is not being used, the same cell also contains the translated code for the procedure which is interpreted. When the procedure returns (or implicitly returns due to an error) the cell is freed again back to the heap. This use of the heap is very tidy - adjacent cells are allocated and freed with little opportunity for leaving gaps in the heap.

Unfortunately, various other keywords also cause cells to be allocated and these can cause fragmentation. For example, LOADM, CREATE, OPEN etc. all allocate cells; UNLOADM, CLOSE etc. free those cells. If a procedure is called which uses CREATE to create a data file, the procedure cell is allocated, followed by the CREATE cell and the procedure cell is then freed when the procedure returns. The heap structure therefore contains a gap where the procedure cell was, which remains until all cells at higher addresses are freed.

❺ On the Series 5, the OPL runtime interpreter uses two separate heaps: one for process management and one for user variables and allocated cells. LOADM, CREATE, OPEN, etc use the process management heap. This ensures that the use of these keywords will not cause fragmentation. It is therefore not necessary for you to understand the interpreter’s use of the heap.

WARNING - PEEKING/POKING THE CELL

Using the allocator is by no means simple in OPL since the data in an allocated cell usually has to be read or written in OPL using the PEEK and POKE set of keywords which are intrinsically subject to programming error. OPL does not provide good support for handling pointers (variables containing addresses), which are basic to heap usage, nor for complicated data structures, so that it is all too easy to make simple programming errors that have disastrous effects.

For these reasons, you are recommended to use the heap accessing keywords only when strictly necessary (which should not be very often) and to take extreme care when you do use them. On the other hand, for programmers with previous experience of dynamic memory allocation, the heap allocation keywords will often prove most useful.

REASONS FOR USING THE HEAP ALLOCATOR

A few common instances where the allocator might be used are:

- when the amount of data to be stored is variable or cannot be determined at the time of writing the program. Without using the allocator, you would have to declare a large array to hold the data always even when it turns out that only a few bytes are needed in a particular case. Using the allocator allows you to grow the cell containing your data as and when required.

- the amount of data may be specified in a file or by the user of the program. Once again, you would need to declare a possibly unnecessarily large array to cope with all allowed cases.

- a system of library procedures might use a common cell, usually called a *control block*, to store common data. You could have one procedure creating the cell and initialising data in it, other procedures in the system could be passed the address of the cell, using and possibly updating the data in it, and finally a further procedure could free the cell.
  This concept will be familiar to you if you have used handles for the I/O keywords, where the handle references a cell used internally by the I/O system.
  If you did not use the allocator in this case, you would probably need to declare a global array in the procedure calling the library procedures, with the disadvantages that the name and size of the array would need to be fixed for all time even when a better alternative mechanism has been devised for the library code with different data requirements.

- ADJUSTALLOC allows you to insert or remove data at the start or in the middle of data that has previously been set up. With an array, you would need to copy each element to the next or previous element to make or close a gap.

USING THE HEAP ALLOCATOR

❺ ALLOC AND ASSOCIATED HEAP KEYWORDS

On the Series 5, ALLOC, REALLOC and ADJUSTALLOC allocate cells that have lengths that are the smallest multiple of four greater than the size requested. All of these raise errors if the cell address argument is not in the range known by the heap. The same address checking is done for peeking and poking, but note that OPL on the Series 5 allows the addresses of the application-specific SIBO magic statics DatApp1 to DatApp7 (hex 28 to 34 inclusive) to be used for compatibility.

ALLOC, REALLOC and ADJUSTALLOC return a long integer value, so that a request can be made to allocate a cell of any length within memory constraints.

See also sections above for discussion of 32-bit addressing.

❸ Note that in the sections which follow, Series 5 32-bit addressing is assumed. If you are using a Series 3c, you can substitute integers for long integers, so for example the usage of ALLOC becomes, `pcell%=ALLOC(size%)`, i.e. `%` rather than `&`. However, using long integers will make any porting to a Series 5 machine in the future easier.

ALLOCATING A CELL

Use `pcell&=ALLOC(size&)` to allocate a cell on the heap of a specified size returning the pointer to the cell or zero if there is not enough memory. **The new cell is uninitialised** - you cannot assume that it is zeroed.

FREEING AN ALLOCATED CELL

Use `FREEALLOC pcell&` to free a previously allocated cell at `pcell&` as returned, for example, by ALLOC. Does nothing if `pcell&` is zero.

CHANGING A CELL'S SIZE

Use `pcelln&=REALLOC(pcell&,size&)` to change the size of a previously allocated cell at address `pcell&` to `size&`, returning the new cell address or zero if there is not enough memory. If out of memory, the old cell at `pcell&` is left as it was.

If successful, `pcelln&` will not be the same as `pcell&` on return only if the size increases and there is no free cell following the cell being grown which is large enough to accommodate the extra amount.

INSERTING OR DELETING DATA IN CELL

Use `pcelln&=ADJUSTALLOC(pcell&,offset&,amount&)` to open or close a gap at `offset&` within the allocated cell `pcell&` returning the new cell address or zero if there is not enough memory. `offset&` is 0 for the first byte in the cell. Opens a gap if `amount&` is positive and closes it if negative. The data in the cell is automatically copied to the new position.

If successful, `pcelln&` will not be the same as `pcell&` on return only if `amount&` is positive and there is no free cell following the cell being adjusted which is large enough to accommodate the extra amount.

FINDING OUT THE CELL LENGTH

Use `len&=LENALLOC(pcell&)` to get the length of the previously allocated cell at `pcell&`.

EXAMPLE USING THE ALLOCATOR

This example illustrates the careful error checking which is essential when using the allocator. RAISE is used to jump to the error recovery code.

If you cannot understand this example it would be wise to avoid using the allocator altogether.

❸ As above, note that Series 5 32-bit addressing is assumed. If you are using a Series 3c, you could substitute integers for long integers, so for example the usage of ALLOC becomes, `pcell%=ALLOC(size%)`, i.e. `%` rather than `&`.

```
LOCAL pcell&                     REM pointer to cell
LOCAL pcelln&                    REM new pointer to cell
LOCAL p&                         REM general pointer
LOCAL n%                         REM general integer
ONERR e1
```

```
pcell&=ALLOC(2+2*8)              REM holds an integer and 2
                                 REM 8-byte floats initially
IF pcell&=0
  RAISE -10                      REM out of memory; go to e1::
ENDIF
POKEW pcell&,2                   REM store integer 2 at start of cell
                                 REM i.e. no. of floats
POKEF UADD(pcell&,2),2.72        REM store float 2.72
POKEF UADD(pcell&,10),3.14       REM store float 3.14 ...
pcelln&=REALLOC(pcell&,2+3*8)    REM space for 3rd float
IF pcelln&=0
  RAISE -10                      REM out of memory
ENDIF
pcell&=pcelln&                   REM use new cell address
n%=PEEKW(pcell&)                 REM no. of floats in cell
POKEF UADD(pcell&,2+n%*8),1.0    REM 1.0 after 3.14
POKEW pcell&,n%+1                REM one more float in cell ...
pcelln&=ADJUSTALLOC(pcell&,2,8)  REM open gap before 2.72
IF pcelln&=0
  RAISE -10                      REM out of memory
ENDIF
pcell&=pcelln&                   REM use new cell address
POKEF UADD(pcell&,2),1.0         REM store 1.0 before 2.72
POKEW pcell&,4                   REM 4 floats in cell now ...
p&=UADD(pcell&,LENALLOC(pcell&)) REM byte after cell end
p&=USUB(p&,8)                    REM address of final float
POKEF p&,90000.1                 REM overwrite with 90000.1
RAISE 0                          REM clear ERR value
e1::
FREEALLOC pcell&                 REM free any cell created
IF err<>0
...                              REM display error message etc.
ENDIF
RETURN ERR
```

INDEX

OPL

USING OPXS ON THE SERIES 5

The Series 5 uses language extensions provided in separate DLLs written specially for OPL support. These DLLs have the file extension OPX.

It is not within the scope of this User Guide to cover how to develop OPXs yourself, as it requires knowledge of the C++ language. If you require details of how to do this, you should obtain a copy of the EPOC32 C++ Software Development Kit (SDK) from Psion Software plc.

OPX procedures for handling the following are provided in the ROM:

- **Date / time extras**
- **System controls**
- **Bitmaps**
- **Sprites**
- **Database extras**
- **Printing**

CONTENTS

OPX OVERVIEW

A maximum of 255 OPXs can be used in any one OPL module and any OPX may have up to 65535 procedures defined in it. If these values are exceeded then errors are reported at translate-time (see the ‘Errors.pdf’ document).

OPX HEADER FILES

The OPX supplier provides an OXH header file. This provides the declaration for the OPX, specifying its name UID and version number (see below), followed by its procedure prototypes.

The new INCLUDE keyword is used to include these header files. Alternatively, the declaration can be inserted directly into the OPL file, i.e.

```
DECLARE OPX <opxName>,<uid>,<version>
   <protoType1> : <ordinal1>
   <protoType2> : <ordinal2>
   ...
END DECLARE
```

where,

`<name>` is the name of the OPX without the .OPX extension. The OPX is stored in a `\System\Opx\` folder on any drive, with the drives scanned from `Y:` to `A:` and then `Z:` if no path is specified. This allows ROM OPXs (in `Z:`) to be overridden if required.

`<uid>` is the UID of the OPX. The specification of a UID as well as a name guards against OPXs with the same name being confused, which could otherwise cause serious problems. The UID is checked on loading the OPX and a ‘Not supported’ error will result if the UIDs in the header file and in the OPX itself do not match.

`<version>` is the version number of the OPX. The version number of an OPX will be increased when any new procedures are added. OPL will refuse to load an OPX which reports that it can’t support the version number given in the declaration. The version number expressed in hex is e.g. $0100 to represent version 1.00, $0102 for version 1.02 etc. In general, an OPX supplier will increment the 2 low digits (the so-called minor version number) for backward compatible changes, and will increment the 2 high digits for major incompatible changes.

`<prototype>` specifies the name, return type and parameters of an OPX procedure in the same way as for an OPL procedure when using the EXTERNAL keyword. Numeric parameters to OPX procedure can be passed *by reference* using BYREF. This means that the value of the variable **can** be changed by the OPX procedure. The OPX procedure prototype is followed additionally by a colon and then the

`<ordinal>` specifies the ordinal number of the procedure in the OPX itself. This is used to call the correct procedure, i.e. the OPX is *linked by ordinal*.

If an OPX procedure name clashes with one of your OPL procedures, or with that of another OPX procedure, you can make a copy of the OXH file and change the name of the offending procedures, but **not** the return type, parameter list or ordinal. You should then include this new OXH file in your module and the new name can then be used to refer to the procedure in your code.

For example, consider the OPX declaration,

```
DECLARE OPX XXOpx,XXOpxUid&,XXOpxVersion&
   XXClose%:(id&) : 1
   ProcWithLongParam:(aLong&) : 2
   AddOneToParams:(BYREF par1%,BYREF par2&, BYREF par3) : 3
END DECLARE
```

If a short integer is passed to `ProcWithLongParam:`, the translator will automatically convert it to a long integer.

The `AddOneToParams:` procedure adds one to each of the parameters. This is possible because the parameters are passed by reference using BYREF. Parameters passed by reference must be variables rather than constant values because the OPX will write back to the variable. The variable type must always be the same as given in the declaration.

CALLBACKS FROM OPX PROCEDURES

An OPX procedure can call back to a module procedure. This is often useful if the OPX needs some information that is not known when the OPX procedure is initially called, or if it requires a large amount of data which needs to be sent piecemeal.

The OPX provider will specify the exact form of the procedure which you must provide.

OPXS INCLUDED IN THE SERIES 5

Procedures for handling the following are provided in the ROM:

- Date / time extras
- System - a variety of procedures for system control, e.g. backlight control, sound control, file and application control, etc.
- Bitmaps - for use with buttons and Sprites.
- Sprites
- Database extras

These OPXs and their OXHs have default paths in the ROM (`Z:`), but the full path for any OPX may be supplied. These are `\System\Opl\` for the header files and `\System\Opx\` for the OPXs themselves. The OPX header files are stored in the ROM, but may be created in RAM by using the ‘Create standard files’ option in the ‘Tools’ menu in the Program editor.

With these OPXs, the OPL programmer is sometimes given direct access to objects via pointers (for efficiency). Otherwise sprites, for example, could not be set up using an array of IDs. These objects can be explicitly deleted to free memory or else they will be deleted when the program exits.

DATE OPX

To use this OPX, you must included the header file Date.oxh, which contains the following declaration:

```
DECLARE OPX DATE,KUidOpxDate&,KOpxDateVersion%
  DTNewDateTime&:(year&,month&,day&,hour&,minute&,second&,micro&) : 1
  DTDeleteDateTime:(id&) : 2
  DTYear&:(id&) : 3
  DTMonth&:(id&) : 4
  DTDay&:(id&) : 5
  DTHour&:(id&) : 6
  DTMinute&:(id&) : 7
  DTSecond&:(id&) : 8
  DTMicro&:(id&) : 9
  DTSetYear:(id&,year&) : 10
  DTSetMonth:(id&,month&) : 11
  DTSetDay:(id&,day&) : 12
  DTSetHour:(id&,hour&) : 13
  DTSetMinute:(id&,minute&) : 14
  DTSetSecond:(id&,second&) : 15
  DTSetMicro:(id&,micro&) : 16
  DTNow&: : 17
  DTDateTimeDiff:(start&,end&,BYREF year&,BYREF month&,BYREF day&,
         BYREF hour&, BYREF minute&, BYREF  second&,BYREF micro&) : 18
  DTYearsDiff&:(st art&,end&) : 19
  DTMonthsDiff&:(start&,end&) : 20
  DTDaysDiff&:(start&,end&) : 21
  DTHoursDiff&:(start&,end&) : 22
  DTMinutesDiff&:(start&,end&) : 23
  DTSecsDiff&:(start&,end&) : 24
  DTMicrosDiff&:(start&,end&) : 25
  DTWeekNoInYear&:(id&,yearstart&,rule&) : 26
  DTDayNoInYear&:(id&,yearstart&) : 27
  DTDayNoInWeek&:(id&) : 28
  DTDaysInMonth&:(id&) : 29
  DTSetHomeTime:(id&) : 30
  LCCountryCode&: : 31
  LCDecimalSeparator$: : 32
  LCSetClockFormat:(format&) : 33
  LCClockFormat&: : 34
  LCStartOfWeek&: : 35
  LCThousandsSeparator$: : 36
END DECLARE
```

DTNEWDATETIME&:

Usage: `id&=DTNEWDATETIME&:(year&,month&,day&,hour&,minute&,second&,micro&)`

Creates a new date/time object which contains all the supplied date/time components and returns a handle `id&` for it.

The year is stored as the usual four figure year, e.g. 1997.

The month is stored as 1 for January, 2 for February, etc.

The day is stored as the day number in the month.

The hour is the hour of the day in 12 or 24 hour clock according to the system setting.

The minutes, seconds and microseconds are stored as the usual values 0 to 59 for minutes and seconds and 0 to 999 for microseconds.

See DTDELETEDATETIME:.

DTDELETEDATETIME:

Usage: `DELETEDATETIME:(id&)`

Deletes the date/time object with handle `id&`. This should be called when a date/time object is no longer needed. Date/time objects will be deleted automatically on unloading the Date OPX or the module which uses it.

See DTNEWDATETIME&:, DTNOW&:.

DTYEAR&:

Usage: `y&=DTYEAR&:(id&)`

Returns the year component `y&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTMONTH&:

Usage: `m&=DTMONTH&:(id&)`

Returns the month component `m&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTDAY&:

Usage: `day&=DTDAY&:(id&)`

Returns the day component `day&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTHOUR&:

Usage: `h&=DTHOUR&:(id&)`

Returns the hour component `h&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:

DTMINUTE&:

Usage: `m&=DTMINUTE&:(id&)`

Returns the minutes component `m&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTSECOND&:

Usage: `s&=DTSECOND&:(id&)`

Returns the seconds component `s&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTMICRO&:

Usage: `m&=DTMICRO&:(id&)`

Returns the microseconds component `m&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTSETYEAR:

Usage: `DTSETYEAR:(y&,id&)`

Sets the year component which is stored in the date/time object with handle `id&` to `y&`.

See DTNEWDATETIME&:.

DTSETMONTH:

Usage: `DTSETMONTH:(m&,id&)`

Sets the month component which is stored in the date/time object with handle `id&` to `m&`.

See DTNEWDATETIME&:.

DTSETDAY:

Usage: `DTSETDAY:(day&,id&)`

Sets the day component which is stored in the date/time object with handle `id&` to `day&`.

See DTNEWDATETIME&:.

DTSETHOUR:

Usage: `DTSETHOUR:(h&,id&)`

Sets the hour component which is stored in the date/time object with handle `id&` to `h&`.

See DTNEWDATETIME&:.

DTSETMINUTE:

Usage: `DTSETMINUTE:(m&,id&)`

Sets the minutes component which is stored in the date/time object with handle `id&` to `m&`.

See DTNEWDATETIME&:.

DTSETSECOND:

Usage: `DTSETSECOND:(s&,id&)`

Sets the seconds component which is stored in the date/time object with handle `id&` to `s&`.

See DTNEWDATETIME&:.

DTSETMICRO:

Usage: `DTSETMICRO:(m&,id&)`

Sets the microseconds component which is stored in the date/time object with handle `id&` to `m&`.

See DTNEWDATETIME&:.

DTNOW&:

Usage: `id&=DTNOW&:`

Creates a new date/time object which contains all the date/time components of the current time and returns a handle `id&` for it.

Example: Timing a loop

```
...
start&=DTNow&:
WHILE condition
...
ENDWH
end&=DTNow&:
PRINT “Time to do loop was”,DTMicrosDiff&:(start&,end&)
...
```

See DTNEWDATETIME&:.

DTDATETIMEDIFF:

Usage: `DTDATETIMEDIFF:(start&,end&,BYREF year&,BYREF month&,BYREF day&,BYREF hour&,BYREF minute&,BYREF second&,BYREF micro&)`

Calculates the exact difference between two date/time objects with handles `start&` and `end&` in the form of a date/time object. The difference is returned in the variables `year&`, `month&` etc.

DTYEARSDIFF&:

Usage: `diff&=DTYEARSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole years between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTMONTHSDIFF&:

Usage: `diff&=DTMONTHSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole months between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTDAYSDIFF&:

Usage: `diff&=DTDAYSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole days between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTHOURSDIFF&:

Usage: `diff&=DTHOURSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole hours between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTMINUTESDIFF&:

Usage: `diff&=DTMINUTESDIFF&:(start&,end&)`

Returns the difference `diff&` in whole minutes between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTSECONDSDIFF&:

Usage: `diff&=DTSECONDSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole seconds between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTMICROSDIFF&:

Usage: `diff&=DTMICROSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole microseconds between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:, DTNOW&:.

DTWEEKNOINYEAR&:

Usage: `w&=DTWEEKNOINYEAR&:(id&,yearstart&,rule&)`

Returns the week number in the year of the date/time object with handle `id&`. The first day of the year is specified by the date/time object with handle `yearstart&`. This would usually be set to 1 January in the appropriate year, but also allows you to set the start of the year to the beginning of the financial year or the academic year, for example.

`rule&` can take three values (0,1,2), allowing the week number to be calculated by one of three different rules:

value	*meaning*
0	the first day of the year is always in week one,
1	requires the first week of the year to have at least four days in it,
2	requires the first week of the year to have the full seven days.

The Agenda application and other Series 5 applications use the rule with value 1 by default.

DTDAYNOINYEAR&:

Usage: `DTDAYNOINYEAR&:(id&,yearstart&)`

Returns the day number in the year of the date/time object with handle `id&`. The first day of the year is specified by the date/time object with handle `yearstart&`.

DTDAYNOINWEEK&:

Usage: `n&=DTDAYNOINWEEK&:(id&)`

Returns the day number in the week (1-7) of the date/time object with handle `id&`.

DTDAYSINMONTH&:

Usage: `n&=DTDAYSINMONTH&:(id&)`

Returns the number of days in the month of the month specified by the month component of the date/time object with handle `id&`.

DTSETHOMETIME:

Usage: `DTSETHOMETIME:(id&)`

Sets the system time to the time specified in the date/time object with handle `id&`.

LCCOUNTRYCODE&:

Usage: `cc&=LCCOUNTRYCODE&:`

Returns the country code for the current system home country (LC stands for ‘Locale’), which may be used to select country-specific data. The country code for any given country is the international dialling prefix for that country

LCDECIMALSEPARATOR$:

Usage: `decSep$=LCDECIMALSEPARATOR$:`

Returns the decimal separator (the character used in decimal numbers to separate whole part from fractional part) according to the local system setting.

LCSETCLOCKFORMAT:

Usage: `LCSETCLOCKFORMAT:(format&)`

Sets the system clock format to either digital or analog. If `format&=0` then the clock is set to analog or if it is 1 then the clock is set to digital.

LCCLOCKFORMAT&:

Usage: `format&=LCCLOCKFORMAT&:`

Returns the current system clock format The procedure returns 0 if the system clock is analog and 1 if it is digital.

LCSTARTOFWEEK&:

Usage: `start&=LCSTARTOFWEEK&:`

Returns the day of the week which set as the first day of the week in the system setting. A return value of 1 indicates Monday, 2 Tuesday, and so on.

LCTHOUSANDSSEPARATOR$:

Usage: `thouSep$=LCTHOUSANDSSEPARATOR$:`

Returns the thousands separator (the character used to separate every three digits of a large number) according to the local system setting.

SYSTEM OPX

To use this OPX, you must included the header file System.oxh, which contains the following declaration:

```
DECLARE OPX SYSTEM,KUidOpxSystem&,KOpxSystemVersion%
   BackLightOn&: : 1
   SetBackLightOn:(state&) : 2
   SetBackLightOnTime:(seconds&) : 3
   SetBacklightBehavior:(behaviour&) : 4
   IsBacklightPresent&: : 5
   SetAutoSwitchOffBehavior:(behaviour&) : 6
   SetAutoSwitchOffTime:(seconds&) : 7
   SetActive:(state&) : 8
   ResetAutoSwitchOffTimer: : 9
   SwitchOff: : 10
   SetSoundEnabled:(state&) : 11
   SetSoundDriverEnabled:(state&) : 12
   SetKeyClickEnabled:(state&) : 13
   SetPointerClickEnabled:(state&) : 14
   SetDisplayContrast:(value&) : 15
   MaxDisplayContrast&: : 16
   IsReadOnly&:(file$) : 17
   IsHidden&:(file$) : 18
   IsSystem&:(file$) : 19
   SetReadOnly:(file$,state&) : 20
   SetHiddenFile:(file$,state&) : 21
   SetSystemFile:(file$,state&) : 22
   VolumeSize&:(drive&) : 23
   VolumeSpaceFree&:(drive&) : 24
   VolumeUniqueID&:(drive&) : 25
   MediaType&:(drive&) : 26
   GetFileTime:(file$,DateTimeId&) : 27
   SetFileTime:(file$,DateTimeId&) : 28
   DisplayTaskList: : 29
   SetComputeMode:(State&) : 30
   RunApp&:(lib$,doc$,tail$,cmd&) : 31
   RunExe&:(name$) : 32
   LogonToThread:(threadId&,BYREF statusWord&) : 33
   TerminateCurrentProcess:(reason&) : 34
   TerminateProcess:(proc$,reason&) : 35
   KillCurrentProcess:(reason&) : 36
   KillProcess:(proc$,reason&) : 37
   PlaySound:(file$,volume&) : 38
   PlaySoundA:(file$,volume&,BYREF statusWord&) : 39
   StopSound&: : 40
   Mod&:(left&,right&) : 41
```

```
  XOR&:(left&,right&) : 42
  LoadRsc&:(file$) : 43
  UnLoadRsc:(id&) : 44
  ReadRsc$:(id&) : 45
  ReadRscLong&:(id&) : 46
  CheckUid$:(Uid1&,Uid2&,Uid3&) : 47
  SetPointerGrabOn:(WinId&,state&) : 48
  MachineName$: : 49
  MachineUniqueId:(BYREF high&,BYREF low&) : 50
  EndTask&:(threadId&,previous&)  : 51
  KillTask&:(threadId&,previous&) : 52
  GetThreadIdFromOpenDoc&:(doc$,BYREF previous&) : 53
  GetThreadIdFromAppUid&:(uid&,BYREF previous&) : 54
  SetForeground: : 55
  SetBackground: : 56
  SetForegroundByThread&:(threadId&,previous&) : 57
  SetBackgroundByThread&:(threadId&,previous&) : 58
  GetNextWindowGroupName$:(threadId&,BYREF previous&) : 59
  GetNextWindowId&:(threadId&,previous&) : 60
  SendKeyEventToApp&:(threadId&,previous&,code&,scanCode&,
  modifiers&,repeats&) : 61
  IrDAConnectToSend&:(protocol$,port&) : 62
  IrDAConnectToReceive:(protocol$,port&,BYREF statusW&) : 63
  IrDAWrite:(chunk$,BYREF statusW&) : 64
  IrDARead$: : 65
  IrDAReadA:(stringAddr&,BYREF statusW&): 66
  IrDAWaitForDisconnect: : 67
  IrDADisconnect: : 68
  MainBatteryStatus&: :69
  BackupBatteryStatus&: :70
  CaptureKey&:(keyCode&,mask&,modifier&) :71
  CancelCaptureKey:(handle&) :72
  SetPointerCapture:(winId&,flags&) :73
  ClaimPointerGrab:(winId&,state&) :74
  OpenFileDialog$:(seedFile$,uid1&,uid2&,uid3&) : 75
  CreateFileDialog$:(seedPath$) : 76
  SaveAsFileDialog$:(seedPath$,BYREF useNewFile%) : 77
END DECLARE
```

BACKLIGHTON&:

Usage: `backLight&=BACKLIGHTON&:`

Returns -1 if the backlight is switched on or 0 if it is switched off.

SETBACKLIGHTON:

Usage: `SETBACKLIGHTON:(state&)`

Switches the backlight on if `state&=1` or off if `state&=0`.

SETBACKLIGHTONTIME:

Usage: `SETBACKLIGHTONTIME:(seconds&)`

Sets the time in seconds (`seconds&`) that the backlight should remain on after it has been switched on.

SETBACKLIGHTBEHAVIOR:

Usage: `SETBACKLIGHTBEHAVIOR:(behavior&)`

Sets the backlight's turning off behaviour.

`behavior&=0` sets the turning off of the backlight to be on a timer,

`behavior&=1` sets the backlight not to be on a timer.

ISBACKLIGHTPRESENT&:

Usage: `ret&=ISBACKLIGHTPRESENT&:`

Returns -1 if there is a backlight present and 0 if there is not.

SETAUTOSWITCHOFFBEHAVIOR:

Usage: `SETAUTOSWITCHOFFBEHAVIOR:(behavior&)`

Sets the machine's auto switch off behaviour.

`behavior&=0` disables the machine's auto switch off mechanism.

`behavior&=1` sets the machine auto switch off to occur only when its batteries are low.

`behavior&=2` sets the machines auto switch off to occur always.

SETAUTOSWITCHOFFTIME:

Usage: `SETAUTOSWITCHOFFTIME:(seconds&)`

Sets the time in seconds (`seconds&`) for which the machine may remain switched on when it is not being used.

SETACTIVE:

Usage: `SETACTIVE:(state&)`

Sets the current process active if `state&=1` or not active if `state&=0`. This will determine whether or not the machine can automatically turn off when the user is not using the machine: if the process is active then the machine will not automatically turn off.

RESETAUTOSWITCHOFFTIMER:

Usage: `RESETAUTOSWITCHOFFTIMER:(seconds&)`

'Tickles' the machine's auto switch off timer, restarting its count down to switching off.

SWITCHOFF:

Usage: `SWITCHOFF:`

Switches off the machine.

⚠ As with the keyword OFF, you should be careful not to use SWITCHOFF: in a loop. If you do, it may be impossible to switch the Series 5 back on, and you may then have to reset it.

SETSOUNDENABLED:

Usage: `SETSOUNDENABLED:(state&)`

Enables the machine’s sound if `state&=1` or disables it if `state&=0`.

SETSOUNDDRIVERENABLED:

Usage: `SETSOUNDDRIVERENABLED:(state&)`

Switches the machines sound driver on if `state&=1` or off if `state&=0`.

SETKEYCLICKENABLED:

Usage: `SETKEYCLICKENABLED:(state&)`

Determines whether or not a keypress makes a click. `state&=1` enables the click and `state&=0` disables it.

SETPOINTERCLICKENABLED:

Usage: `SETPOINTERCLICKENABLED:(state&)`

Determines whether or not a pointer event makes a click. (A pointer event occurs whenever the machine’s screen is pressed.) `state&=1` enables the click and `state&=0` disables it.

SETDISPLAYCONTRAST:

Usage: `SETDISPLAYCONTRAST:(value&)`

Sets the contrast on the machine’s screen. `value&` specifies the contrast value, which can be between zero and the maximum display contrast.

See MAXDISPLAYCONTRAST&:.

MAXDISPLAYCONTRAST&:

Usage: `maxContrast&=MAXDISPLAYCONTRAST&:`

Returns the maximum value to which the machines display contrast can be set.

See SETDISPLAYCONTRAST:.

ISREADONLY&:

Usage: `readOnly&=ISREADONLY&:(file$)`

Returns -1 if the file, `file$`, is a read-only file and 0 if it is not.

ISHIDDEN&:

Usage: `hidden&=ISHIDDEN&:(file$)`

Returns -1 if the file, `file$`, is hidden by the system and 0 if it is not.

ISSYTEM&:

Usage: `system&=ISSYTEM&:(file$)`

Returns -1 if the file, `file$`, is a system file and 0 if it is not.

SETREADONLY:

Usage: `SETREADONLY:(file$,state&)`

Sets the file, `file$`, to be read only if `state&=1` or not read-only if `state&=0`.

SETHIDDENFILE:

Usage: `SETHIDDENFILE:(file$,state&)`

Sets the file, `file$`, to be hidden by the system if `state&=1` or not to be hidden if `state&=0`.

SETSYSTEMFILE:

Usage: `SETSYSTEMFILE:(file$,state&)`

Sets the file, `file$`, to be a system file if `state&=1` or not to be a system file if `state&=0`.

VOLUMESIZE&:

Usage: `VOLUMESIZE&:(drive&)`

Returns the size in bytes of the storage space on the drive specified by `drive&`. `drive&` can take values 0 to 25. 0 specifies the `A:`, 1 specifies `B:`, 2 specifies `C:` and so on.

VOLUMESPACEFREE&:

Usage: `free&=VOLUMESPACEFREE&:(drive&)`

Returns the size in bytes of the storage space that is available for use on the drive specified by `drive&`. `drive&` can take values 0 to 25. 0 specifies the `A:`, 1 specifies `B:`, 2 specifies `C:` and so on.

VOLUMEUNIQUEID&:

Usage: `volUid&=VOLUMEUNIQUEID&:(drive&)`

Returns the unique identification number of the media on the drive specified by `drive&`. `drive&` can take values 0 to 25. 0 specifies the `A:`, 1 specifies `B:`, 2 specifies `C:` and so on.

See MEDIATYPE&:.

MEDIATYPE&:

Usage: `media&=MEDIATYPE&:(drive&)`

Returns the media type present on the drive specified by `drive&`. `drive&` can take values 0 to 25. 0 specifies the `A:`, 1 specifies `B:`, 2 specifies `C:` and so on. The value returned may be any of the following,

value	*meaning*	*constant declaration in System.oxh*
0	no media present	`CONST KMediaNotPresent&=0`
1	media unknown	`CONST KMediaUnknown&=1`
2	floppy disk	`CONST KMediaFloppy&=2`
3	hard disk	`CONST KMediaHardDisk&=3`
4	CD-ROM	`CONST KMediaCdRom&=4`
5	RAM	`CONST KMediaRam&=5`
6	flash	`CONST KMediaFlash&=6`
7	ROM	`CONST KMediaRom&=7`
8	remote	`CONST KMediaRemote&=8`

GETFILETIME:

Usage: `GETFILETIME:(file$,dateTimeId&)`

Returns the time that the file, `file$`, was last modified into `dateTimeId&`. It is necessary to pass this procedure the ID of a date/time object which the procedure will then set to the required time. To obtain and read a date/time object, see the ‘Date OPX’ section.

SETFILETIME:

Usage: `SETFILETIME:(file$,dateTimeId&)`

Sets the time that the file, `file$`, was last modified to `dateTimeId&`. It is necessary to pass this procedure the ID of a date/time object which the procedure will use to set the time. To get a date/time object, which provides microsecond accuracy, see the ‘Date OPX’ section.

DISPLAYTASKLIST:

Usage: `DISPLAYTASKLIST:`

Displays the system-wide task list. The user may then close a file, go to another task or close the dialog.

SETCOMPUTEMODE:

Usage: `SETCOMPUTEMODE:(state&)`

Changes the priority control for the current program.

The following values are available for `state&`:

value	*meaning*	*constant declaration in System.oxh*
`0`	compute mode disabled	`CONST KComputeModeDisabled&=0`
`1`	compute mode on	`CONST KComputeModeOn&=1`
`2`	compute mode off	`CONST KComputeModeOff&=2`

This puts the current process into compute mode (`state&=KComputeModeOn&`) or takes it out of compute mode (`state&=KComputeModeOff&`). In compute mode, a process runs at the lower background priority even when it is the foreground process. Disabling compute mode control (`state&=KComputeModeDisabled&`) prevents the window server from changing the program's priority when it moves between background and foreground.

OPL runs in compute mode by default to support simple OPOs which cannot always be assumed to be well-behaved programs.

This default behaviour is necessary because OPL programs would not otherwise give any background programs a chance to run, simply by running in a tight loop. If your program doesn't run in a tight loop, i.e. if it calls GETEVENT32, GET, etc. regularly, you can use `SETCOMPUTEMODE:(KComputeModeOff&)`.

Note that TBarInit: in Toolbar.opo sets compute mode off, since any program that has a toolbar shouldn't be running in a tight loop at any time. (See the 'Friendlier Interaction' section of the 'GUI.pdf' document for more details.)

RUNAPP&:

Usage: `thread&=RUNAPP&:(lib$,doc$,tail$,cmd&)`

Runs an application and returns a thread ID for the application. The thread ID can be used to logon to the application, to find out when and why it finished. This ID can also be used to locate the window group, end the task etc.

`lib$` is the C++ application name.

`doc$` is the document name, if any, to pass to the application.

`tail$` is the tail end, needed only by certain applications such as OPL.

`cmd&` can take values:

value	*meaning*
0	open
1	create
2	run
3	background

The values 0, 1 and 2 are the same as for `CMD$(3)` (see the 'Alphabetic Listing' section of the 'Glossary.pdf' document). The value 3 will run the application in the background.

For example, to run an OPL program:

```
k&=RUNAPP&:(“OPL”,“”,“RZ:\System\Opl\Toolbar.opo”,2)
```

Note that `tail$` contains a leading `R`: this is required by OPL.

To open the Data help file:

```
k&=RUNAPP&:(“Data”,“Z:\System\Data\Help”,“”,0)
```

See also the 'OPL Applications' section in the 'Advanced.pdf' document.

See LOGONTOTHREAD:.

RUNEXE&:

Usage: `RUNEXE&:(file$)`

Runs an executable EXE file `file$` and returns its thread ID. The thread ID can then be used to logon to the thread and find out when and why it finished.

See LOGONTOTHREAD:.

LOGONTOTHREAD:

Usage: `LOGONTOTHREAD:(threadId&,statusWord&)`

Logs on to the thread with ID `threadId&` and sets the status word `statusWord&` when the thread has completed. As for other 32-bit status words, `statusW&` will be set to &80000001 for pending and 0 for successful completion.

TERMINATECURRENTPROCESS:

Usage: `TERMINATEPROCESS:(reason&)`

Terminates the current process giving it the reason `reason&`. This causes the process to receive a shutdown message. `reason&` may take any value, with zero used to mean no errors.

TERMINATEPROCESS:

Usage: `TERMINATEPROCESS:(proc$,reason&)`

Terminates the process `proc$` giving it the reason `reason&`. This causes the process to receive a shutdown message. `reason&` may take any value, with zero used to mean no errors.

KILLCURRENTPROCESS:

Usage: `KILLCURRENTPROCESS:(reason&)`

Kills the current process giving it the reason `reason&`. The process is killed **without** receiving a shutdown message. `reason&` may take any value, with zero used to mean no errors.

KILLPROCESS:

Usage: `KILLPROCESS:(proc$,reason&)`

Kills the process `proc$` giving it the reason `reason&`. The process is killed **without** receiving a shutdown message. `reason&` may take any value, with zero used to mean no errors.

PLAYSOUND:

Usage: `PLAYSOUND:(file$,volume&)`

Plays the file `file$`, which may be a sound file or an alarm file. Does not return until the sound has completed.

`volume&` specifies the volume at which the file should be played, 0 specifies no volume and 3 specifies maximum volume. The play will be synchronous (i.e. the OPL program will stop running until the file has finished playing). For asynchronous sound playing see PLAYSOUNDA:.

PLAYSOUNDA:

Usage: `PLAYSOUNDA:(file$,volume&,statusWord&)`

Plays the file `file$`, which may be a sound file or an alarm file, asynchronously. It returns immediately, with the status word being set on completion or cancellation of the sound.

`volume&` specifies the volume at which the file should be played, 0 specifies no volume and 3 specifies maximum volume. The play will be asynchronous (i.e. the OPL program will continue running while the sound file is playing). `statusWord&` is the variable which will contain information about the state of the sounds file play. For synchronous sound playing see PLAYSOUND:.

STOPSOUND&:

Usage: `STOPSOUND&:`

Stops the sound file that is currently playing. The return value is 1 if there was a sound file playing and 0 if not.

See PLAYSOUND:, PLAYSOUNDA:.

MOD&:

Usage: `rem&=MOD&:(left&,right&)`

Returns `left&` modulo `right&` (i.e. the remainder of `left&` divided by `right&`).

XOR&:

Usage: `res&=XOR&:(left&,right&)`

Returns the exclusive OR of `left&` and `right&`. A bit in the result has value 1 if the corresponding bit is set in **either** `left&` or in `right&`, but **not in both**, otherwise it is 0.

E.g. with `left&=5` (binary 00000101) and `right&=3` (binary 00000011), `XOR&:(left&,right&)` returns 6 (binary 00000110).

`left&`	00000101
`right&`	00000011
`XOR&:(left&,right&)`	00000110

This is often used to invert particular bits in an integer. So `left&=XOR&:(left&,3)` inverts bits 0 and 1 in `left&`, where bit 0 is the rightmost bit, and leaves the other bits alone.

LOADRSC&:

Usage: `id&=LOADRSC&:(file$)`

Loads the resource file `file$` and returns a handle for it.

See UNLOADRSC:, READRSC$:.

UNLOADRSC:

Usage: `UNLOADRSC:(id&)`

Unloads the resource file whose handle is `id&`.

See LOADRSC&:.

READRSC$:

Usage: `string$=READRSC$:(id&)`

Reads the resource in a resource file specified by `id&` which must already be loaded.

See LOADRSC&:.

READRSCLONG&:

Usage: `long&=READRSCLONG&:(id&)`

Reads a 32-bit value from a resource file specified by the `id&` which must already be loaded.

See LOADRSC&:.

CHECKUID$:

Usage: `CHECKUID$:(Uid1&,Uid2&,Uid3&)`

Returns a string which is a unique product of the three supplied UIDs.

SETPOINTERGRABON:

Usage: `SETPOINTERGRABON:(WinId&,state&)`

Enables (or disables) pointer grabs in a window. After this function has been called with `state&=1`, any “down” event in this window will cause a pointer grab, terminated by the next corresponding “up” event. The terminating “up” event is also sent to the window with the grab.

This function would typically be used for “drag-and-drop”, and would typically be called after window creation so that pointer grab is enabled for the lifetime of the window.

`state&=0` disables the grab.

See CLAIMPOINTERGRAB:.

MACHINENAME$:

Usage: `name$=MACHINENAME$:`

Returns the machine name.

MACHINEUNIQUEID:

Usage: `MACHINEUNIQUEID:(BYREF high&,BYREF low&)`

Sets `high&` and `low&` (which are passed BYREF) to high and low parts of the machine’s unique ID.

ENDTASK&:

Usage: `ret&=ENDTASK&:(threadId&,previous&)`

Sends a *shutdown* message to a window group in the thread, `threadId&`.

`previous&` specifies the window group in the thread that is previous to the one required. Hence, setting `previous&` to 0 will specify the first window group in the thread.

The value returned by this procedure is the value of the window group on which action was taken, or -1 if a window group following that specified by `previous&` was not present. This return value could be passed back to this procedure to end the next window group in the thread.

An error will be raised if the window group is busy, does not respond to such requests or is in a system thread.

See KILLTASK&:.

KILLTASK&:

Usage: `ret&=KILLTASK&:(threadId&,previous&)`

Shuts down the window group that follows the window group specified by `previous&` in the thread, `threadId&`. It will ignore any wishes of the window group not to be shutdown.

The value returned by this procedure is the value of the window group on which action was taken, or -1 if a window group following that specified by `previous&` was not present. This return value could be passed back to this procedure to end the next window group in the thread.

See ENDTASK&:.

GETTHREADIDFROMOPENDOC&:

Usage: `threadId&=GETTHREADIDFROMOPENDOC&:(doc$,BYREF previous&)`

Returns the thread ID `threadId&` of the thread that contains the open document `doc$`.

`doc$` should contain the full path of the open document and could for example be `c:\opl\prog.opo`.

`previous&`, passed by reference, is useful for situations where the document may be open more than once. Setting `previous&` to zero will cause this procedure to find the first window group that contains `doc$`. If one is found, `previous&` will be set to the window group value of that found document. This value could then passed back to this procedure as `previous&`, so the next window group instance of `doc$` will be found, and so on. If this procedure sets `previous&` to -1 then `doc$` could not be found after `previous&`. An error will also be raised if the open document is not found.

GETTHREADIDFROMAPPUID&:

Usage: `threadId&=GETTHREADIDFROMAPPUID&:(uid&,BYREF previous&)`

Returns the thread ID `threadId&` of the thread that contains a running instance of the application with UID `uid&`.

`previous&`, passed by reference, is useful for situations where the application may be running more than once. Setting `previous&` to zero will cause this procedure to find the first window group that is an instance of this application. If one is found, `previous&` will be set to the value of that window group. This value could then passed back to this procedure as `previous&`, so the next instance of the application can be found, and so on. If no instance of this application can be found following `previous&` then `previous&` will be set to -1. An error will also be raised if a running instance of the application is not found.

SETFOREGROUND:

Usage: `SETFOREGROUND:`

Moves the calling process to foreground.

See SETBACKGROUND:.

SETBACKGROUND:

Usage: `SETBACKGROUND:`

Moves the calling process to background.

See SETFOREGROUND:.

SETFOREGROUNDBYTHREAD&:

Usage: `ret&=SETFOREGROUNDBYTHREAD&:(threadId&,previous&)`

Moves the window group after `previous&` in the thread `threadId&` to foreground. Using `previous&=0` specifies the first window group in the thread.

The value returned will be the window group on which action was taken, or -1 if the following window group was not found. This return value could be passed back to this procedure as `previous&` to put the next window group in `threadId&` to foreground.

See SETBACKGROUNDBYTHREAD&:.

SETBACKGROUNDBYTHREAD&:

Usage: `ret&=SETBACKGROUNDBYTHREAD&:(threadId&,previous&)`

Moves the window group after `previous&` in the thread `threadId&` to background. Using `previous&=0` specifies the first window group in the thread.

The value returned will be the window group on which action was taken, or -1 if the following window group was not found. This return value could be passed back to this procedure again as a value for `previous&` to put the next window group in `threadId&` to background.

See SETFOREGROUNDBYTHREAD&:.

GETNEXTWINDOWGROUPNAME$:

Usage: `name$=GETNEXTWINDOWGROUPNAME$:(threadId&,BYREF previous&)`

Returns the name of the window group that follows the window group `previous&` in the thread `threadId&`. If `previous&=0` the procedure will return the name of the first window group found.

See GETNEXTWINDOWID&:.

GETNEXTWINDOWID&:

Usage: `winId&=GETNEXTWINDOWID&:(threadId&,BYREF previous&)`

Returns the ID of the window group that follows the window group `previous&` in the thread `threadId&`. If `previous&` is 0 the procedure will return the ID of the first window group found.

See GETNEXTWINDOWGROUPNAME$:.

SENDKEYEVENTTOAPP&:

Usage: `SENDKEYEVENTTOAPP&:(threadId&,previous&,code&,scanCode&,modifiers&,repeats&)`

Sends a key event to the window group (application) that follows the window group with ID `previous&` (or the first window group if `previous&=0)`, in the thread with ID `threadId&`. The key event is specified by `code&`, `scanCode&`, `modifiers&` and `repeats&`.

The value returned is the window group to which the key was sent.

IRDACONNECTTOSEND&:

Usage: `IRDACONNECTTOSEND&:(protocol$,port&)`

Synchronously sends out an infrared (IR) beam which looks for another infrared device. If another infrared device is looking to receive a beam then the devices will connect, otherwise this procedure will raise an error after a couple of seconds.

`protocol$` can take either of the constants defined in System.oxh. These are `KIrTinyTP$`, used for communicating with other EPOC32 devices and `KIrmux$` which is used for IR printing.

The value 8 is recommended for `port&` when communicating with other EPOC32 devices. Both devices will need to be using the same value for `port&`. `port&` must be 2 for connecting to IR printers.

If connecting to another EPOC32 device with this procedure the other device must connect by using IRDACONNECTTORECEIVE&:.

After connecting to another EPOC32 device you can both send and receive information.

After connecting to an IR printer use IRDAWRITE: to send text. When IRDADISCONNECT: is called the printer will begin printing.

See IRDACONNECTTORECEIVE:, IRDAWRITE:, IRDAREAD:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDACONNECTTORECEIVE:

Usage: `IRDACONNECTTORECEIVE:(protocol$,port&,BYREF statusW&)`

Asynchronously waits, without timing-out, for another EPOC32 device to attempt to connect via infrared (IR). This procedure takes the status word `statusW&`. When connection occurs the status word is set to zero.

Both EPOC32 devices must be using the same port which is specified by `port&`. The recommended port is 8.

One of the devices must connect with this procedure and the other with IRDACONNECTTOSEND&.

`protocol$` specifies the IR protocol and should take the constant `KIrTinyTP$`, defined in System.oxh, when communicating with other EPOC32 devices.

After connecting to another EPOC32 device you can both send and receive information.

See IRDAWRITE:, IRDAREAD:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDAWRITE:

Usage: `IRDAWRITE:(chunk$, BYREF statusW&)`

Sends a string `chunk$` via infrared to another device after connection has been made. The procedure is asynchronous and the success of sending the string is reported in `statusW&`.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDAREAD;, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDAREAD$:

Usage: `chunk$=IRDAREAD$:`

Reads a text string via infrared from another device after connection has been made. The procedure is synchronous.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDWRITE:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDAREADA:

Usage: `IRDAREADA:(stringAddr&, BYREF statusW&)`

Reads a text string via infrared from another device after connection has been made. The procedure is asynchronous with status word `statusW&`. The string passed by address should have a maximum length of 255 bytes. To pass a string by address use, `ADDR(string$)`.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDWRITE:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDAWAITFORDISCONNECT:

Usage: `IRDAWAITFORDISCONNECT:`

Verifies, when disconnecting from another IR device, that the other device has received everything sent. It should therefore be called by the device that last sends some information.

This procedure should not be used with printers: use IRDADISCONNECT: instead.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDADISCONNECT:

Usage: `IRDADISCONNECT:`

Disconnects from another IR device. It should be called by the device that is the last to receive some information. IRDAWAITFORDISCONNECT: is used by the last device to send some information, except when printing via IR when IRDADISCONNECT: should be used to disconnect from the printer and commence printing.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

MAINBATTERYSTATUS&:

Usage: `MAINBATTERYSTATUS&:`

Returns the current power of the main batteries. The following possible return values are supplied as constants in System.oxh:

```
CONST KBatteryZero&          = 0
CONST KBatteryVeryLow&       = 1
CONST KBatteryLow&           = 2
CONST KBatteryGood&          = 3
```

BACKUPBATTERYSTATUS&:

Usage: `BACKUPBATTERYSTATUS&:`

Returns the current power of the backup batteries. The following possible return values are supplied as constants in System.oxh:

```
CONST KBatteryZero&            = 0

CONST KBatteryVeryLow&         = 1

CONST KBatteryLow&             = 2

CONST KBatteryGood&            = 3
```

CAPTUREKEY&:

Usage: `CAPTUREKEY&:(keyCode&,mask&,modifier&)`

Allows the program that calls it to capture keys when it is not in foreground. The keys to be captured are specified using `keycode&`, `mask&` and `modifier&`. The value returned is a handle which can be passed to CANCELCAPTUREKEY: to cancel the capture.

The following constants are supplied in System.oxh to be used in conjunction with this procedure:

```
CONST KModifierAutorepeatable&  = &00000001
CONST KModifierKeypad&          = &00000002
CONST KModifierLeftAlt&         = &00000004
CONST KModifierRightAlt&        = &00000008
CONST KModifierAlt&             = &00000010
CONST KModifierLeftCtrl&        = &00000020
CONST KModifierRightCtrl&       = &00000040
CONST KModifierCtrl&            = &00000080
CONST KModifierLeftShift&       = &00000100
CONST KModifierRightShift&      = &00000200
CONST KModifierShift&           = &00000400
CONST KModifierLeftFunc&        = &00000800
CONST KModifierRightFunc&       = &00001000
CONST KModifierFunc&            = &00002000
CONST KModifierCapsLock&        = &00004000
CONST KModifierNumLock&         = &00008000
CONST KModifierScrollLock&      = &00010000
CONST KModifierKeyUp&           = &00020000
CONST KModifierSpecial&         = &00040000
CONST KModifierDoubleClick&     = &00080000
CONST KModifierPureKeycode&     = &00100000
CONST KAllModifiers&            = &001fffff
```

See CANCELCAPTUREKEY&:.

CANCELCAPTUREKEY:

Usage: `CANCELCAPTUREKEY:(handle&)`

Cancels an outstanding key capture which is specified by the handle `handle&`.

See CAPTUREKEY&:.

SETPOINTERCAPTURE:

Usage: `SETPOINTERCAPTURE:(WinId&,flags&)`

Sets the (OPL) window with window ID `winId&` to capture pointer events.

`flags&` can take a summed combination of the following:

value	*meaning*
0	capture disabled
&01	capture enabled
&02	capture (not drag-and-drop)

When a window captures pointer events, the position of any events received will be relative to its origin, i.e. the top left corner of that window, i.e. the window's 0,0 co-ordinate becomes the origin for pointer events. Events don't need to be inside the window to be registered. For example, if the screen was touched 1 pixel above and 1 pixel to the left of a window that was capturing pointer events, the pointer event would be returned with the position of the event being -1,-1 .

"Capture (not drag-and-drop)" will cause events of the "drag-and-drop" nature to be sent to the window that would have received them had the pointer not been captured.

CLAIMPOINTERGRAB:

Usage: `CLAIMPOINTERGRAB:(WinId&,state&)`

Claims the pointer grab for the window with ID `winId&` from another window, if a pointer grab is already in effect in the other window. All subsequent events will be delivered to this window, instead of the window that originally had the pointer grab. The next "up" event terminates the pointer grab, and is also delivered to the window that claimed the grab, if `state&` is set to 1.

This function would typically be used where clicking in a window pops up another window, and where the popped-up window wishes to grab the pointer as though the original click had been in that window.

To summarise the values of `state&` for the OPL window,

value	*meaning*
0	don't deliver the following "up" event to this window
1	deliver the following "up" event to this window

See SETPOINTERGRABON:

OPENFILEDIALOG$:

Usage: `file$=OPENFILEDIALOG$:(seedFile$,uid1&,uid2&,uid3&)`

Displays the standard 'Open file' dialog. `seedFile$` provides a starting file which is displayed when the dialog is opened. So, for example, if `seedFile$="C:\Documents\Myfile"` then the file `MyFile` will be initially displayed in the 'Name' selector box, the `Documents` folder will be initially displayed in the 'Folder' selector box and `C` will be displayed in the disk selector. You must always seed the dialog: you cannot pass a null `seedFile$` string to OPENFILEDIALOG$:. If `seedFile$` does not include a drive name, then an error is raised.

The selection of files may be restricted by UID by specifying appropriate values for `uid1&`, `uid2&` and `uid3&` in exactly the same way as when using the dFILE keyword. See the 'Alphabetic Listing' section for details of this.

The return value is the filename, including the full path of the file selected to be opened.

CREATEFILEDIALOG$:

Usage: `file$=CREATEFILEDIALOG$:(seedPath$)`

Displays the standard 'Create new file' dialog. `seedPath$` provides a starting path which is displayed when the dialog is opened. So, for example, if `seedPath$="C:\Documents\"` then the `Documents` folder will be intially displayed in the 'Folder' selector box and `C` in the disk selector. If you supply a filename in `seedPath$`, then this will be displayed as a suggested filename in the 'Name' selector box. You must always seed the dialog: you cannot pass a null `seedPath$` string to CREATEFILEDIALOG$:. If `seedPath$` does not include a drive name, then an error is raised.

The return value is the filename, including the full path of the file to be created.

SAVEASFILEDIALOG$:

Usage: `path$=SAVEASFILEDIALOG$:(seedPath$,BYREF useNewFile%)`

Displays the standard 'Save as' dialog. `seedPath$` provides a starting path which is displayed when the dialog is opened. So, for example, if `seedPath$="C:\Documents\"` then the `Documents` folder will be initially displayed in the 'Folder' selector box and `C` in the disk selector. If you supply a filename in `seedPath$`, then this will be displayed as a suggested filename in the 'Name' selector box. You must always seed the dialog: you cannot pass a null `seedPath$` string to SAVEASFILEDIALOG$:. If `seedPath$` does not include a drive name, then an error is raised.

`UseNewFile%` specifies the initial setting of the checkbox which determines whether a new file should be used when saving. This value is non-zero then the tick will be set initially, if it is zero, then it will not be. The procedure writes back a value to this variable which will be `KTrue%` if the symbol is set on exiting the dialog and `KFalse%` if not (see the listing of Const.oph in Appendix E in the 'Appends.pdf' document for definitions of these constants). If you pass #0 as the value of this argument, then this item will not be displayed in the dialog at all (as in many Series 5 applications) so that whether a new file is used on not is not decided by the user, but by the programmer.

The return value is the filename to save as, including the full path.

SPRITE AND BITMAP OPX

In general sprite procedures behave in a similar way to the sprite keywords of the Series 3a, 3c and Siena.

To use this OPX, you must included the header file Bmp.oxh, which contains the following declaration:

```
DECLARE OPX BMP,&10000258,$100
   BITMAPLOAD&:(name$,index&) : 1
   BITMAPUNLOAD:(id&) : 2
   BITMAPDISPLAYMODE&:(id&) : 3
   SPRITECREATE&:(winId%,x&,y&,flags&) : 4
   SPRITEAPPEND:(time&,bitmap&,maskBitmap&,invertMask&,dx&,dy&) : 5
   SPRITECHANGE:(index&,time&,bitmap&,maskBitmap&,invertMask&,dx&, dy&) : 6
   SPRITEDRAW:() : 7
   SPRITEPOS:(x&,y&) : 8
   SPRITEDELETE:(id&) : 9
   SPRITEUSE:(id&) : 10
END DECLARE
```

BITMAPLOAD&:

Usage: `id&=BITMAPLOAD&:(name$,index&)`

Loads a bitmap from the file `name$` (in the current directory if no other is specified). If the file contains more than one bitmap, then `index&` specifies which bitmap to load. The first (or only) bitmap is specified by `index&` having the value 0. The value returned is the ID of the open bitmap, which may be used to refer to it when calling sprite procedures or when using gBUTTON, for example.

See BITMAPUNLOAD:.

BITMAPUNLOAD:

Usage: `BITMAPUNLOAD:(id&)`

Unloads the bitmap whose ID is `id&`. You can call BITMAPUNLOAD: immediately after passing the bitmap ID to gBUTTON or, if used with the Sprite OPX procedures, **after** drawing the sprite, as the access count on a bitmap is incremented to ensure it is not unloaded prematurely.

See BITMAPLOAD&:.

BITMAPDISPLAYMODE&:

Usage: `mode&=BITMAPDISPLAYMODE&:(id&)`

Returns the graphics mode of the bitmap with ID `id&`. The graphics mode is that specified at its creation (see gCREATEBIT), i.e. 0 for 2-colour mode, 1 for 4-colour mode and 2 for 16-colour mode.

SPRITECREATE&:

Usage: `id&=SPRITECREATE&:(winId%,x&,y&,flags&)`

Initialises a sprite in the window given by `winId%` with its top left-hand corner at `x&,y&`. The value of `flags&` determines whether or not the sprite’s members are flashing. If `(flags& AND 1)=1` then they are flashing and if the value is 0 then they are not. The value returned is the ID of the sprite which should be used when referring to it in other sprite procedures.

See SPRITEAPPEND:, SPRITECHANGE:, SPRITEDRAW:, SPRITEPOS:, SPRITEDELETE:, SPRITEUSE:.

SPRITEAPPEND:

Usage: `SPRITEAPPEND:(time&,bitmap&,maskBitmap&,invertMask&,dx&,dy&)`

Appends *members* or *frames* to the current sprite, which must have been created using CREATESPRITE&: before attempting to append to it. `bitmap&` is the ID of the bitmap to be used for this member of the sprite, and `maskBitmap&` is the ID of the bitmap to be used as a mask. The bitmap and mask must be of identical sizes. `invertMask&` takes the value of 1 or 0 to determine whether the mask is to be inverted or not respectively. For example, if you are using identical bitmap and mask and `invertMask&=1` then the member will appear as the bitmap, whereas if `invertMask&=0`, the member will be blank. `time&` is the period of time in microseconds for which the member appears in the sprite and `dx&,dy&` is the offset position of the top left-hand of the bitmap from the top left-hand corner of the sprite.

See SPRITECHANGE:.

SPRITECHANGE:

Usage: `SPRITECHANGE:(id&,time&,bitmap&,maskBitmap&,invertMask&,dx&,dy&)`

Changes a member of a sprite originally set up with SPRITEAPPEND.

`id&` specifies the number of the sprite member which is to be changed. This number is determined by the order in which the members were originally appended, the first member to be appended being labelled 0. `bitmap&` is the ID of the bitmap to be used for this member of the sprite, and `maskBitmap&` is the ID of the bitmap to be used as a mask. The bitmap and mask must be of identical sizes. `invertMask&` takes the value of 1 or 0 to determine whether the mask is to be inverted or not respectively. For example, if you are using identical bitmap and mask and `invertMask&=1` then the member will appear as the bitmap, whereas if `invertMask&=0`, the member will be blank. `time&` is the period of time in microseconds for which the member appears in the sprite and `dx&,dy&` is the offset position of the top left-hand of the bitmap from the top left-hand corner of the sprite.

A sprite may not be changed unless it has already been drawn.

See SPRITEAPPEND:, SPRITEDRAW:.

SPRITEDRAW:

Usage: `SPRITEDRAW:`

Draws the sprite once it has been set up. It need only be called once, and any changes made to the sprite are made as soon as the procedure making the change is called. The sprite will be drawn in the window and at the position specified by the arguments passed to SPRITECREATE.

See SPRITECREATE&:, SPRITEAPPEND:, SPRITECHANGE:, SPRITEPOS:.

SPRITEPOS:

Usage: `SPRITEPOS:(x&,y&)`

Repositions the whole of the current sprite to the point `x&,y&`.

See SPRITECREATE&:,SPRITEDRAW:.

SPRITEDELETE:

Usage: `SPRITEDELETE:(id&)`

Deletes the sprite with ID `id&`.

SPRITEUSE:

Usage: `SPRITEUSE:(id&)`

Sets the current sprite to be that with ID `id&` in order that it may be changed.

See SPRITECHANGE:, SPRITEPOS:.

✎ Note that using many sprites at one time may cause undesirable effects. Firstly, sprites maybe animated at a slower speed than requested, and secondly the response time to any key or pointer event can be lengthened considerably. The number of sprites which can be drawn at one time without these effects occurring depends mainly on the animation speed: the faster it is the less sprites may be used successfully at one time. As a general rule, it is advisable not to use more than around 20 sprites at any one time when the animation speed is 0.2 sec, but you should experiment to find out what is suitable for your particular program.

DATABASE OPX

To use this OPX, you must included the header file Dbase.oxh, which contains the following declaration:

```
DECLARE OPX DBASE,KUidOpxDBase&,KOpxDBaseVersion%
  DbAddField:(keyPtr&,fieldName$,order&) : 1
  DbAddFieldTrunc:(keyPtr&,fieldName$,order&,trunc&) : 2
  DbCreateIndex:(index$,keyPtr&,dbase$,table$) : 3
  DbDeleteKey:(keyPtr&) : 4
  DbDropIndex:(index$,dbase$,table$) : 5
  DbGetFieldCount&:(dbase$,table$) : 6
  DbGetFieldName$:(dbase$,table$,fieldNum&) : 7
  DbGetFieldType&:(dbase$,table$,fieldNum&) : 8
  DbIsDamaged&:(dbase$) : 9
  DbIsUnique&:(keyPtr&) : 10
  DbMakeUnique:(keyPtr&) : 11
  DbNewKey&: : 12
  DbRecover:(dbase$) : 13
  DbSetComparison:(KeyPtr&,comp&) : 14
END DECLARE
```

DBADDFIELD:

Usage: `DBADDFIELD:(key&,field$,order&)`

Adds the field, `field$`, to the key, `key&` (see DBNEWKEY&: for creating a new key). This new key can then be used to create an index on a table. `order&` specifies how this field should be ordered:

value	*meaning*	*constant declaration in Dbase.oxh*
1	ascending	`CONST KDbAscending&     = 1`
0	descending	`CONST KDbDescending&    = 1`

You can use this procedure repeatedly to add more than one field to the same key. The order in which the fields are added dictates the significance of these fields in building up the index. The first field added to a key is the primary field; if there were any identical values in this field then the secondary field would be used and so on. For example, if the primary field in an ‘Address’ table was ‘Surname’ and there were two people with the surname Brown, then the secondary field or tertiary fields like ‘Forename’ and ‘Age’ might be used to define how the index will be ordered.

When using string fields in an index they cannot be longer than 240 bytes. To limit the size of a string field see CREATE in the ‘Alphabetic Listing’ section of the ‘Glossary.pdf’ document. If a string field is the last field in a key (or first and only) then it may be truncated to a specified length - see DBADDFIELDTRUNC:.

Note that the size of an index can become huge if the key is long. The key length *k* is the length in bytes of the sum of all the fields in the key. An integer or a long integer field is 4 bytes, a floating-point field 8 bytes and a text field depends on the length assigned to it (see also the previous paragraph). The size of the index depends at least linearly on *k* and hence may be large if the key is long.

When the key has been built as required, it can be used to create an index.

See DBCREATEINDEX:.

DBADDFIELDTRUNC:

Usage: `DBADDFIELDTRUNC:(key&,field$,order&,trunc&)`

Adds a truncated string field `field$` to the key `key&`. Only the last field added to a key (or first and only) can be truncated. `order&` specifies how this field should be ordered and may take the values:

value	*meaning*	*constant declaration in Dbase.oxh*
1	ascending	`CONST KDbAscending&     = 1`
0	descending	`CONST KDbDescending&    = 1`

`trunc&` is the truncation length and determines how many bytes of the string field will be used in building up the index (up to 240).

This procedure is useful for building up indexes on OPL16 databases whose string fields have a set length of 255 bytes. When the key has been built up as required it can be used to create an index.

See DBADDFIELD:, DBCREATEINDEX:.

DBCREATEINDEX:

Usage: `DBCREATEINDEX:(index$,key&,file$,table$)`

Creates an index with the name `index$` on the table `table$` in the database `file$`. The index is used for sorting and vastly increases the speed in table lookup. The index is based on `key&`, the supplied key. The database must be closed when this procedure is used.

The opening of an ordered view on a table must be requested in the SQL query in the OPEN command. See Appendix F in the ‘Appends.pdf’ document for SQL specification. If a suitable index has been created then it will be used.

See DBNEWKEY:, DBADDFIELD:, DBADDFIELDTRUNC:, DBMAKEUNIQUE:, DBSETCOMPARISON:, DBDROPINDEX:, OPEN.

DBDELETEKEY:

Usage: `DBDELETEKEY:(key&)`

Deletes the key `key&`. This should be called as soon as the key is no longer required. Any keys left undeleted when all modules that declare or include a declaration of the Dbase OPX have been unloaded will be automatically deleted

See DBNEWKEY&.

DBDROPINDEX:

Usage: `DBDROPINDEX:(index$,file$,table$)`

Drops the index `index$` of the table `table$` of the database `file$`. The database must be closed to use this procedure.

See DBCREATEINDEX:.

DBGETFIELDCOUNT&:

Usage: `n&=DBGETFIELDCOUNT&:(dbase$,table$)`

Returns the number of fields in one of the records in the table `table$` of the database `dbase$`. This number can then be used to analyse the contents of the record.

See DBGETFIELDNAME$:, DBGETFIELDTYPE&:.

DBGETFIELDNAME$:

Usage: `name$=DBGETFIELDNAME$:(dbase$,table$,fieldNum&)`

Returns the name of the field whose number is `fieldNum&` in the table `table$` of the database `dbase$`. This is useful for analysing the records of a table.

See DBGETFIELDCOUNT&:, DBGETFIELDTYPE&:.

DBGETFIELDTYPE&:

Usage: `type&=DBGETFIELDTYPE&:(dbase$,table$,fieldNum&)`

Returns the type of the field whose number is `fieldNum&` in the table `table$` of the database `dbase$`. This is useful for analysing the records of a table.

The values that may be returned (many of which aren't supported by OPL databases), are as follows:

value	*type*
0	bit
1	signed byte (8 bits)
1	unsigned byte (8 bits)
2	integer (16 bits)
3	unsigned integer (16 bits)
4	long integer (32 bits)
5	unsigned long integer (32 bits)
6	64-bit integer
7	single precision floating-point number (32 bits)
8	double precision floating-point number (64 bits)
9	date/time object
10	ASCII text
11	Unicode text
12	Binary
13	LongText8
14	LongText16
15	LongBinary

Types 2,4,8,10 correspond to OPL's `%`,`&`, floating point and `$` types respectively.

See DBGETFIELDCOUNT&, DBGETFIELDNAME&

DBISDAMAGED&:

Usage: `i&=DbIsDamaged&:(dbase$)`

Returns 1 if the database `dbase$` considers that it may have damaged indexes and 0 if not. If the database is damaged, then this does not make it unusable, but attempting to use any damaged index will result in an error. Recovering the database will restore any damaged indexes.

See DBRECOVER:.

DBISUNIQUE&:

Usage: `i&=DBISUNIQUE&:(key&)`

Returns -1 if the key `key&` is unique or 0 if it is not.

See DBMAKEUNIQUE:.

DBMAKEUNIQUE:

Usage: `DBMAKEUNIQUE:(key&)`

Sets the key `key&` to be unique. The index created will then not allow any records to be added if they exactly match the indexed fields of another record.

See DBNEWKEY:, DBADDFIELD:, DBCREATEINDEX:.

DBNEWKEY&:

Usage: `k&=NEWKEY&:`

Returns a handle to a key which can be used for creating indexes.

See DBDELETEKEY:, DBCREATEINDEX:, DBADDFIELD:, DBADDFIELDTRUNC:.

DBRECOVER:

Usage: `DBRECOVER:(dbase$)`

Recovers a damaged database `dbase$`, restoring any damaged indices.

See DBISDAMAGED&:.

DBSETCOMPARISON:

Usage: `DBSETCOMPARISON:(key&,comp&)`

The text comparison is used to determine the order of an index. The argument `comp&` sets the text comparison mode of a key `key&`, and takes one of the constant values supplied in the header file Dbase.oxh:

```
CONST KDbCompareNormal&      = 0
CONST KDbCompareFolded &     = 1
CONST KDbCompareCollated&    = 2
```

PRINTER OPX

Printer OPX provides access to print and text handling. An object to be printed is first created dynamically by sending text, bitmaps and formatting information. Printing itself is then handled by calling standard print dialogs. One procedure is provided for each of the four dialogs usually called from standard applications.

```
DECLARE OPX PRINTER,KUidOpxPrinter&,KOpxPrinterVersion%
   SendStringToPrinter:(string$) : 1
   InsertString:(string$,pos&) : 2
   SendNewParaToPrinter: : 3
   InsertNewPara:(pos&) : 4
   SendSpecialCharToPrinter:(character%) : 5
   InsertSpecialChar:(character%,pos&) : 6
   SetAlignment:(alignment%) : 7
   InitialiseParaFormat:(Red&, Green&, Blue&, LeftMarginInTwips&,
                 RightMarginInTwips&, IndentInTwips&,
                 HorizontalAlignment%, VerticalAlignment%,
                 LineSpacingInTwips&, LineSpacingControl%,
                 SpaceBeforeInTwips&, SpaceAfterInTwips&, KeepTogether%,
                 KeepWithNext%, StartNewPage%, WidowOrphan%, Wrap%,
                 BorderMarginInTwips&, DefaultTabWidthInTwips&) : 8
   SetLocalParaFormat: : 9
   SetGlobalParaFormat: : 10
   RemoveSpecificParaFormat: : 11
   SetFontName:(name$) : 12
   SetFontHeight:(height%) : 13
   SetFontPosition:(pos%) : 14
   SetFontWeight:(weight%) : 15
   SetFontPosture:(posture%) : 16
   SetFontStrikethrough:(strikethrough%) : 17
   SetFontUnderline:(underline%) : 18
   SetGlobalCharFormat: : 19
   RemoveSpecificCharFormat: : 20
   SendBitmapToPrinter:(bitmapHandle&) : 21
   InsertBitmap:(bitmapHandle&,pos&) : 22
   SendScaledBitmapToPrinter:(bitmapHandle&,xScale&,yScale&) : 23
   InsertScaledBitmap:(bitmapHandle&,pos&,xScale&,yScale&) : 24
   PrinterDocLength&: : 25
   SendRichTextToPrinter:(richTextAddress&) : 26
   ResetPrinting: : 27
   PageSetupDialog: : 28
   PrintPreviewDialog: : 29
   PrintRangeDialog: : 30
   PrintDialog: : 31
   SendBufferToPrinter:(Addr&) : 32
END DECLARE
```

SENDSTRINGTOPRINTER:

Usage: `SENDSTRINGTOPRINTER:(string$)`

Append a string to whatever has already been sent to the printer.

See INSERTSTRING:, SENDNEWPARATOPRINTER:.

INSERTSTRING:

Usage: `INSERTSTRING:(string$,pos&)`

Insert `string$` at position `pos&` in the buffer. Inserting at position zero puts the string ahead of anything already sent or inserted.

See SENDSTRINGTOPRINTER:, INSERTNEWPARA:.

SENDNEWPARATOPRINTER:

Usage: `SENDNEWPARATOPRINTER:`

Paragraphs delimit paragraph formatting, such as centring. This procedure is equivalent to

`SENDSPECIALCHARTOPRINTER:(KParagraphDelimiter%).`

See SENDSPECIALCHARTOPRINTER:, INSERTNEWPARA:, SENDSTRINGTOPRINTER:.

INSERTNEWPARA:

Usage: `INSERTNEWPARA:(pos&)`

Inserts a new paragraph at position `pos&` in the buffer.

See SENDNEWPARATOPRINTER:, INSERTSTRING:.

SENDSPECIALCHARTOPRINTER:

Usage: `SENDSPECIALCHARTOPRINTER:(character%)`

Sends a special character, for example line and page breaks etc., to the printer. Constants for special characters are defined 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) as follows:

```
CONST KParagraphDelimiter%    = $06
CONST KLineBreak%             = $07
CONST KPageBreak%             = $08
CONST KTabCharacter%          = $09
CONST KNonBreakingTab%        = $0a
CONST KNonBreakingHyphen%     = $0b
CONST KPotentialHyphen%       = $0c
CONST KNonBreakingSpace%      = $10
CONST KVisibleSpaceCharacter%= $0f
```

See SENDNEWPARATOPRINTER:, INSERTSPECIALCHAR:.

INSERTSPECIALCHAR:

Usage: `INSERTSPECIALCHAR:(character%,pos&)`

Inserts a special character at `pos&` in the buffer.

See SENDSPECIALCHARTOPRINTER:.

SETALIGNMENT:

Usage: `SETALIGNMENT:(alignment%)`

Sets alignment state of a paragraph. Default is `KPrintLeftAlign%`.

The setting does not take effect until either SETLOCALPARAFORMAT: or SETGLOBALPARAFORMAT: is called. The allowable alignments, defined in Printer.oxh, are:

```
CONST KPrintLeftAlign%        = 0
CONST KPrintCenterAlign%      = 1
CONST KPrintRightAlign%       = 2
CONST KPrintJustifiedAlign%   = 3
CONST KPrintUnspecifiedAlign%= 4
```

See INITIALISEPARAFORMAT:.

INITIALISEPARAFORMAT:

Usage:
```
INITIALISEPARAFORMAT:(Red%, Green%, Blue%, LeftMarginInTwips&,
                      RightMarginInTwips&, IndentInTwips&,
                      HorizontalAlignment%, VerticalAlignment%,
                      LineSpacingInTwips&, LineSpacingControl%,
                      SpaceBeforeInTwips&, SpaceAfterInTwips&,
                      KeepTogether%, KeepWithNext%, StartNewPage%,
                      WidowOrphan%, Wrap%, BorderMarginInTwips&,
                      DefaultTabWidthInTwips&)
```

Sets a state for formatting. The setting does not take effect until either `SETLOCALPARAFORMAT` or `SETGLOBALPARAFORMAT` is called.

`Red%,Green%,Blue%` set the background colour. Each value can be in the range 0 to 255. Default colour is white, with all three arguments equal to 255.

`LeftMarginInTwips&` sets left text margin relative to left page margin. Default is zero.

`RightMarginInTwips&` sets right text margin, relative to right page margin. Default is zero.

`IndentInTwips&` sets left, right and first line indent. Default is zero.

`HorizontalAlignment%` sets horizontal alignment of paragraph. Default is left alignment. See `SETALIGNMENT` for values.

`VerticalAlignment%` sets vertical alignment of paragraph (for use by spreadsheet applications). Default is top alignment. Allowable values, supplied in Printer.oxh, are:

```
CONST KPrintTopAlign%          = 0

CONST KPrintBottomAlign%       = 2

CONST KPrintUnspecifiedAlign%  = 4
```

`LineSpacingInTwips&` sets inter-line spacing in twips. Default is 200 (10 point).

`LineSpacingControl%` sets the control for `LineSpacingInTwips&` value. Default is `KLineSpacingAtLeastInTwips&`. Allowable values are:

```
CONST KLineSpacingAtLeastInTwips%= 0

CONST KLineSpacingExactlyInTwips%= 1
```

`SpaceBeforeInTwips&` sets the space above a paragraph. Default is zero.

`SpaceAfterInTwips&` sets the space below a paragraph. Default is zero.

`KeepTogether%` prevents a page break within paragraph when `KTrue%`. Default is `KFalse%`. (Constants are defined in Const.oph.)

`KeepWithNext%` prevents a page break between this paragraph and the following paragraph when `KTrue%`. Default is `KFalse%`.

`StartNewPage%` inserts a page break immediately before this paragraph when `KTrue%`. Default is `KFalse%`.

`WidowOrphan%` prevents the printing of the last line of a paragraph by itself on the top of a new page (widow) or the first line of a paragraph by itself on the bottom of the page (orphan). Default is `KFalse%`.

`Wrap%` forces the paragraph to line wrap at the right margin when `KTrue%`. `KFalse%` disables line wrap. Default is `KTrue%`.

`BorderMarginInTwips&` sets distance in twips between paragraph border and enclosed text (must be non-negative). Default is zero.

`DefaultTabWidthInTwips&` specifies the spacing between the default tab stops. Default is 360.

SETLOCALPARAFORMAT:

Usage: `SETLOCALPARAFORMAT:`

Sets the initialised global paragraph formatting as local.

See SETGLOBALPARAFORMAT:.

SETGLOBALPARAFORMAT:

Usage: `SETGLOBALPARAFORMAT:`

Sets the local paragraph format as global (after initialising formatting).

See SETLOCALPARAFORMAT:, REMOVESPECIFICPARAFORMAT:.

REMOVESPECIFICPARAFORMAT:

Usage: `REMOVESPECIFICPARAFORMAT:`

Unsets local paragraph formatting, using global formatting instead

See SETGLOBALPARAFORMAT:, SETLOCALPARAFORMAT:.

SETFONTNAME:

Usage: `SETFONTNAME:(name$)`

Gives a new font name. Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTHEIGHT:

Usage: `SETFONTHEIGHT:(height%)`

Gives a new font height in twips

(1 twip = 1/20 points or 1/1440 inches)

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTPOSITION:

Usage: `SETFONTPOSITION:(POS%)`

Sets font position as normal, superscript or subscript. The following constants can be used for this:

```
CONST KPrintPosNormal%        = 0
CONST KPrintPosSuperscript%   = 1
CONST KPrintPosSubscript%     = 2
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTWEIGHT:

Usage: `SETFONTWEIGHT:(weight%)`

Sets the font weight (normal or bold). Allowable values are:

```
CONST KStrokeWeightNormal%    = 0
CONST KStrokeWeightBold%      = 1
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTPOSTURE:

Usage: `SETFONTPOSTURE:(posture%)`

Sets font posture using the following constants:

```
CONST KPostureUpright%        = 0
CONST KPostureItalic%         = 1
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTSTRIKETHROUGH:

Usage: `SETFONTSTRIKETHROUGH:(strikethrough%)`

Set strike-through on or off using:

```
CONST KStrikethroughOff%      = 0
CONST KStrikethroughOn%       = 1
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTUNDERLINE:

Usage: `SETFONTUNDERLINE:(underline%)`

Set underline on or off using:

```
CONST KUnderlineOff%          = 0
CONST KUnderlineOn%           = 1
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETGLOBALCHARFORMAT:

Usage: `SETGLOBALCHARFORMAT:`

Sets the currently active character format to be the global default.

See REMOVESPECIFICCHARFORMAT:.

REMOVESPECIFICCHARFORMAT:

Usage: `REMOVESPECIFICCHARFORMAT:`

Unsets local character formatting, replacing it with the global formatting instead.

See SETGLOBALCHARFORMAT:.

SENDBITMAPTOPRINTER:

Usage: `SENDBITMAPTOPRINTER:(bitmapHandle&)`

Appends a bitmap from its handle `bitmapHandle&`.

See INSERTBITMAP:, SENDSCALEDBITMAPTOPRINTER:.

INSERTBITMAP:

Usage: `INSERTBITMAP:(bitmapHandle&,pos&)`

Inserts a bitmap specified by `bitmapHandle&` at position `pos&` in the buffer to be sent to the printer. The handle is the same as that returned by gLOADBIT.

See SENDBITMAPTOPRINTER:, INSERTSCALEDBITMAP:.

SENDSCALEDBITMAPTOPRINTER:

Usage: `SENDSCALEDBITMAPTOPRINTER:(bitmapHandle&,xScale&,yScale&)`

Appends a scaled bitmap specified by `bitmapHandle&` and scaled according to `xScale&` and `yScale&` to the buffer sent to the printer. The handle is the same as that returned by gLOADBIT. Default scaling is `xScale& = yScale& = 1000`.

See SENDBITMAPTOPRINTER:, INSERTSCALEDBITMAP:.

INSERTSCALEDBITMAP:

Usage: `INSERTSCALEDBITMAP:(bitmapHandle&,pos&,xScale&,yScale&)`

Inserts a scaled bitmap specified by `bitmapHandle&` and scaled according to `xScale&` and `yScale&` at position `pos&` in the buffer sent to the printer. The handle is the same as that returned by gLOADBIT.

See INSERTBITMAP:, SENDSCALEDBITMAPTOPRINTER:.

PRINTERDOCLENGTH&:

Usage: `PRINTERDOCLENGTH&:`

Returns the count of characters currently held in the buffer. It is not possible to insert characters beyond the length of the document buffer. Bitmaps and special characters each count as one character.

SENDRICHTEXTTOPRINTER:

Usage: `SENDRICHTEXTTOPRINTER:(richTextAddress&)`

Sends the address of a rich text object `richTextAddress&` to the printer. This is intended to be used on a pointer returned by another OPX. This new Rich Text will **replace** all content currently stored.

RESETPRINTING:

Usage: `RESETPRINTING:`

Deletes **all** text, bitmaps and formatting in the printing buffer.

PAGESETUPDIALOG:

Usage: `PAGESETUPDIALOG:`

Calls the standard page setup dialog.

See PRINTPREVIEWDIALOG:, PRINTRANGEDIALOG:, PRINTDIALOG:.

PRINTPREVIEWDIALOG:

Usage: `PRINTPREVIEWDIALOG:`

Calls the standard print preview dialog. This allows the data sent to the printer to be viewed. The other three dialogs can all be called from this dialog.

See PAGESETUPDIALOG:, PRINTRANGEDIALOG:, PRINTDIALOG:.

PRINTRANGEDIALOG:

Usage: `PRINTRANGEDIALOG:`

Calls a standard print range dialog.

See PAGESETUPDIALOG:, PRINTPREVIEWDIALOG:, PRINTDIALOG:.

PRINTDIALOG:

Usage: `PRINTDIALOG:`

Calls a standard print dialog.

See PAGESETUPDIALOG:, PRINTPREVIEWDIALOG:, PRINTRANGEDIALOG:.

SENDBUFFERTOPRINTER:

Usage: `SENDBUFFERTOPRINTER:(addr&)`

Appends the contents of a buffer to whatever has already been sent to the printer. The `addr&` parameter should be the address of a buffer into which text has been inserted by dEDITMULTI.

INDEX

OPL

OVERVIEW
&
ALPHABETIC LISTING OF COMMANDS

CONTENTS

OVERVIEW

Keywords can be subdivided into *functions*, which return a value, and *commands*, which do not. In practice you use functions and commands together, often using functions as if they were commands, ignoring the values they return.

This section lists all the keywords, grouped according to their purpose. Use this section if you know what you'd like to do, but not which function or command will do it.

The section of this document which follows this one lists the keywords alphabetically, with explanations and full specifications.

PROGRAM CONTROL

LOOPS, BRANCHES, JUMPS

Repeat a set of instructions	DO...UNTIL, WHILE...ENDW
Do one set of instructions or another, or another, or another...	IF...ENDIF
Go...	
...to a specified label	GOTO
...to one of a list of labels	VECTOR…ENDV
...to the end/start of a repeating set of instructions	BREAK, CONTINUE
...back to the calling procedure	RETURN
End the program	STOP

ERROR HANDLING

Raise an error	RAISE
Put an explanatory comment in your program	REM
Declare an error-handler	ONERR…ONERR OFF
Let the program continue after an error	TRAP
After an error, find out what the error was	ERR, ERR$
❺ After an error, find out what the extended error message was	ERRX$

SCREEN AND KEYBOARD CONTROL

Display a string to be edited and get a value from the keyboard	EDIT
Get a value from the keyboard	INPUT
Display text, numbers etc.	PRINT
Set screen update method	gUPDATE
Pause...	
...for a number of seconds	PAUSE
...until a key is pressed	GET, GET$
Position or hide the cursor	AT, CURSOR
Clear the text window	CLS
Sound the buzzer	BEEP
Set the size/position of the text window	SCREEN
Get information on the text window	SCREENINFO
Set text window font and style	FONT, STYLE
Find out which key was pressed, if any	KEY, KEY$, GET, GET$

Find out what combination of modifiers was pressed	KMOD
Disable/enable stopping from a running program	ESCAPE OFF/ON
Turn the Psion off	OFF

FILES

GENERAL FILE MANAGEMENT

Copy a file	COPY
Delete or rename a file	DELETE, RENAME
Check to see if a certain file exists	EXIST
Find out what files there are	DIR$

OPL PROCEDURES AND MODULES

❸ Set up a procedure cache	CACHE
Load an OPL module file so you can use the procedures in it	LOADM
Remove a module from memory	UNLOADM
❺ Include a file containing constant definitions and procedure prototypes	INCLUDE
❺ Cause a error to be raised by the translator if a procedure or variable is used before it is declared	DECLARE EXTERNAL
❺ Declare a procedure or variable as external	EXTERNAL

DATA FILES AND DATABASES

Create a new data file, database or table	CREATE
OPEN or CLOSE a data file, database or table	OPEN, OPENR, CLOSE
❺ Delete a table from a database	DELETE
Use a different data file that has been opened	USE
❸ Copy a data file, optionally appending to another data file, and removing deleted records	COMPRESS

Once a data file has been OPENed, you can:

Make a new record	APPEND
Change a record	UPDATE

Search for those records which contain a certain string	FIND, FINDFIELD
Erase a record	ERASE
Move to a different record	FIRST, LAST, NEXT, BACK, POSITION
Count the records	COUNT
Find whether you’re at the end of the file yet	EOF
Find the current record number	POS
❸ Find the number of bytes used by the current record	RECSIZE
❺ Begin a transaction	BEGINTRANS
❺ Commit a transaction	COMMITTRANS
❺ Find out whether the current view is in transaction	INTRANS
❺ Cancel the transaction	ROLLBACK
❺ Insert a bookmark	BOOKMARK
❺ Go to a bookmark	GOTOMARK
❺ Delete a bookmark	KILLMARK
❺ Insert a new blank record	INSERT
❺ Modify a record	MODIFY
❺ Make changes to a database permanent	PUT
❺ Cancel changes made to a database	CANCEL
❺ Compact a database (see COMPRESS)	COMPACT

MANAGING DIRECTORIES

Create directory	MKDIR
Set current directory	SETPATH
Remove directory	RMDIR

MEMORY

Declare variables	GLOBAL, LOCAL
❺ Declare a constant	CONST
Find how much free memory there is on a device	SPACE

PRINTING

Specify a device or file to print to	LOPEN
Close the print device or file opened with LOPEN	LCLOSE
Print to a device or file	LPRINT

NUMBERS

TRIGONOMETRY

Trig functions	COS, SIN, TAN, ACOS, ASIN, ATAN
Convert between degrees and radians	RAD, DEG

OTHER FUNCTIONS

Raise e to a power	EXP
Logarithms	LN, LOG
Pi as a constant	PI
Square root	SQR
Use random numbers	RND, RANDOMIZE
Unsigned integer/pointer arithmetic (in the 64K limit)	UADD, USUB

LISTS OF NUMBERS

Find the greatest or smallest value in the list	MAX, MIN
Average the list	MEAN
Add up the list	SUM
Find the standard deviation or variance	STD, VAR

CHANGING THE FORMAT OF NUMBERS

Knock the minus sign off a number	ABS, IABS
Take whole number, removing any fractional part	INT, INTF
Convert…	
...an integer into floating-point	FLT
...an integer into a hexadecimal string	HEX$
...a number into a string	FIX$, GEN$, SCI$, NUM$
...a string into a number	EVAL, VAL

STRINGS

Copy characters from a string	LEFT$, MID$, RIGHT$
Repeat a string	REPT$
Make a string upper or lower case	UPPER$, LOWER$

Find out...	
...how long a string is	LEN
...the character code of the first character of a string	ASC
...where a certain string is within a string	LOC
Convert...	
...a string of digits to a number	VAL
...a number to a string	FIX$, GEN$, SCI$, NUM$
Get the character with a certain character code	CHR$

DATE AND TIME

Find out the current date and time...	
...as a string	DATIM$
...just the current time	SECOND, MINUTE, HOUR
...just the current date	DAY, MONTH, YEAR
Find out...	
...the number of days between two dates	DAYS
...what day of the week, or week number, a certain date falls in	DOW, WEEK
Express...	
...1-12 as the name of a month	MONTH$
...1-7 as a day of the week	DAYNAME$
Convert between seconds and dates	DATETOSECS, SECSTODATE
❺ Convert between days and dates	DAYSTODATE

GRAPHICS

DRAWING COMMANDS

Set current position	gAT, gMOVE
❺ Set current pen width	gSETPENWIDTH
Draw a line	gLINEBY, gLINETO
Draw a sequence of lines	gPOLY
❺ Draw a circle	gCIRCLE
❺ Draw an ellipse	gELLIPSE
Draw a rectangle	gBOX
Draw a border	gBORDER, gXBORDER

Fill a rectangle	gFILL
Invert a rectangle	gINVERT
Scroll a rectangle	gSCROLL
Get current position	gX, gY
Display a running clock	gCLOCK
Draw a 3-D button (key)	gBUTTON
❸ Draw a lozenge	gDRAWOBJECT

DISPLAYING GRAPHICS TEXT

Display a list of expressions	gPRINT
Display text in a cleared box	gPRINTB
Display text neatly clipped	gPRINTCLIP
Find width required by text	gTWIDTH
Display text underlined/highlighted	gXPRINT

SETTING STYLES

Set graphics to set / clear / invert pixels	gGMODE
Set text to set / clear / invert / replace pixels	gTMODE
Set font to use	gFONT
Load and unload user-defined fonts	gLOADFONT, gUNLOADFONT
Set text to bold / underline / inverse / double / mono / italic	gSTYLE

WINDOWS AND BITMAPS

Create a new window	gCREATE
Set position and/or size of a window	gSETWIN
Set order to show windows	gORDER
Get order of a window	gRANK
Set window visible / invisible	gVISIBLE
Get screen position of a window	gORIGINX, gORIGINY
Create a bitmap	gCREATEBIT
Load a bitmap from file	gLOADBIT
Clear a window / bitmap	gCLS
Save window / bitmap to bitmap file	gSAVEBIT
Close down a window / bitmap	gCLOSE
Set which window / bitmap to use	gUSE

❺ Set foreground colour of current window	gCOLOR
❺ Swap between grey and black pens	gGREY
❸ Set grey on/off in a window	gGREY
❺ Set colour mode of default window	DEFAULTWIN
❸ Enable grey in default window	DEFAULTWIN
Fill an area with repetitions of another window / bitmap	gPATT
Copy an area from one window / bitmap to another	gCOPY
Read data back from a window / bitmap	gPEEKLINE
Get ID number of a window / bitmap	gIDENTITY
Get size of a window / bitmap	gWIDTH, gHEIGHT
❸ Get status information about a drawable and about the cursor	gINFO
❺ Get status information about a drawable and about the cursor	gINFO32

❸ SPRITES

Create a sprite	CREATESPRITE
Define bitmap-sets for a sprite	APPENDSPRITE, CHANGESPRITE
Draw a sprite	DRAWSPRITE
Set a sprite’s position	POSSPRITE
Close a sprite	CLOSESPRITE

MENUS

Start a new set of menus	mINIT
Define a menu	mCARD
❺ Define a cascade for a menu	mCASC
Display menus	MENU
❺ Define a popup menu	mPOPUP

DIALOGS

Start a new dialog	dINIT
Position a dialog	dPOSITION
Define text for a dialog	dTEXT
Define an edit box for a dialog	dEDIT
❺ Defines a multi-line edit box for a dialog	dEDITMULTI
Define a secret edit box for a dialog	dXINPUT
Define a filename edit box for a dialog	dFILE
Define a choice list for a dialog	dCHOICE
❺ Defines an item with a checkbox for a dialog	dCHECKBOX
Define a numeric edit box for a dialog	dFLOAT, dLONG
Define a date or time edit box for a dialog	dDATE, dTIME
Define exit keys for a dialog	dBUTTONS
Display a dialog	DIALOG
Display a simple ‘alert’ dialog	ALERT

SERIES 3C AND SIENA STATUS WINDOW

Display/hide status window	STATUSWIN
Get status window information	STATWININFO
Set a program’s name	SETNAME
Initialise a diamond list	DIAMINIT
Position the diamond symbol on a diamond list	DIAMPOS

SCREEN MESSAGES

Display information messages	GIPRINT
Display ‘busy’ messages	BUSY

OPL

OPL APPLICATIONS

Define an OPA	APP…ENDA
Declares an OPAs icon	ICON
❺ Give an OPAs caption in the given language	CAPTION
❺ Specify whether an OPA is document-based	FLAGS
❸ Set the type of an OPA	TYPE
❸ Give the file extension used by an OPA	EXT
❸ Give the directory to use for an OPA’s files	PATH
Mark an OPA as locked or unlocked	LOCK

ADVANCED USE

❸ Run machine code	USR, USR$
Find out where a certain variable is in memory	ADDR
Store a value in a specific place in memory	POKE commands
Find out the value stored at a certain place in memory	PEEK commands
Open any type of file	IOOPEN
Read from a file opened with IOOPEN	IOREAD
Write to a file opened with IOOPEN	IOWRITE
Close a file opened with IOOPEN	IOCLOSE
Position within a file opened with IOOPEN	IOSEEK
Keywords which provide low-level access to the Psion	
❸ Call an operating system service	CALL, OS
Perform an asynchronous I/O function	IOA, IOC
Cancel an asynchronous I/O function	IOCANCEL
Wait for completion of a function performed by IOA or IOC	IOWAIT, IOWAITSTAT,
❺ Wait for completion an asynchronous OPX procedure	IOWAITSTAT32
Signal completion of an I/O function	IOSIGNAL
Ensure an asynchronous handler runs	IOYIELD
Perform a synchronous I/O function	IOW
Perform an asynchronous keyboard read	KEYA

OPL

Cancel a KEYA	KEYC
Get command line information	CMD$, GETCMD$
❺ Set a file to be a document	SETDOC
❺ Get the name of the current document	GETDOC$
Parse a full file specification	PARSE$
Check for system events	GETEVENT, TESTEVENT
❺ Check for system and pointer events	GETEVENT32, GETEVENT32A
❺ Filter pointer events out or reinstate them	POINTERFILTER
❸ Get system-level info on data files	ODBINFO
❸ Load/link a DYL	LOADLIB, LINKLIB
❸ Unload a DYL	UNLOADLIB
❸ Find category handles	FINDLIB, GETLIBH
❸ Create new objects	NEWOBJ, NEWOBJH
❸ Send a message to an object	SEND, ENTERSEND, ENTERSEND0
Allocate a heap cell	ALLOC
Free an allocated cell	FREEALLOC
Change size of allocated cell	REALLOC
Insert or delete section of cell	ADJUSTALLOC
Find length of allocated cell	LENALLOC
❸ Remove returned procedures from a cache	CACHETIDY
❸ Read cache index header	CACHEHDR
❸ Read cache index record	CACHEREC
❺ Set flags (mainly for Series 3c compatibility)	SETFLAGS
❺ Clears flags set by SETFLAGS	CLEARFLAGS

ALPHABETIC LISTING OF COMMANDS

This section explains how keywords (functions and commands) are specified and used, then lists them all alphabetically. Use this section if you know which keyword you need to use, but need to check how to use it. Each one is listed with the specification of its usage, then a description of what it does.

Note that the example programs in this section do not include full error handling code. This means that the programs have been kept short and easy to understand, but may fail if, for example, you enter the wrong type of value for a variable. If you want to develop programs from these examples, it is recommended that you add some error handling code to them. See the ‘Errors.pdf’ document.

TYPING COMMANDS, FUNCTIONS AND ARGUMENTS

- Commands, functions and arguments may be typed in any combination of UPPER and lower case.
- To put more than one statement on a line, separate them by a space followed by a colon - e.g.

  ```
  CLS :PRINT “hello” :GET
  ```

  Any commands may be strung together like this, and as many of them as you like, provided the total line length does not exceed 255 characters. The colon is optional before a REM statement.
- Where one space is allowed, any number of spaces is allowed, e.g.

  ```
  CLS   : PRINT “Press Esc”
  ```
- Functions may be used as arguments to other functions or commands e.g. `PRINT LEFT$(A$,3)` and `a=COS(ABS(x))` are OK.

HOW COMMANDS ARE SPECIFIED

Commands are specified as,

```
COMMAND argument(s)
```

where `argument(s)` follow the command **after a space** and are **separated from each other by commas**. The arguments may include:

- Floating-point expression (e.g. `SIN(30)+2`), variable (e.g. `price` or `z`) or literal value (e.g. `78.9`) (or declared constant (e.g. `KFixed`) on the Series 5)
- Integer expression (e.g. `3*567`), variable (e.g. `price%`, or `price&` if in range) or literal value (e.g. `-5676`) (or declared constant (e.g. `KFixed%`) on the Series 5)
- Long integer expression (e.g. `3*56799`), variable (e.g. `profit&`) or literal value (e.g. `-5676869`) (or declared constant (e.g. `KFixed&`) on the Series 5)
- String expression (e.g. `b$+MID$(a$)`), variable (e.g. `price$`) or literal value (e.g. `“word”`) (or declared constant (e.g. `KFixed$`) on the Series 5)
- Logical file name (`A-Z` on the Series 5; `A`, `B`, `C` or `D` on the Series 3c)
- Field name

For example, `AT X%,Y%` might be used like this: `AT 15,2`

HOW FUNCTIONS ARE SPECIFIED

Functions are specified as,

```
variable=FUNCTION(argument(s))
```

where `variable` may be `f%` or `f&` for a function returning an integer or long integer result, `f` for a function returning a floating-point result, or `f$` for a function returning a string result. The `argument(s)`:

- follow the command immediately
- are enclosed in brackets ( )

- are separated from each other in the brackets by a comma
- may include variables, literal values or expressions (or declared constants on the Series 5) of the appropriate kind - integer, long integer, floating-point or string, as described above.

E.g. `f$=LEFT$(g$,x%)` might be used like this: `PRINT LEFT$(fname$,2)`

If you use the wrong type of number as an argument it will, where possible, be converted. For example, you can use an integer for a floating-point argument, or a long integer for an integer argument. If the conversion is not possible - for example, if you use a floating-point number for an integer argument and its value is outside the range of integers an error will be produced and the program stopped. Some functions, such as GET, have no arguments.

COMMANDS

ABS

Usage: `a=ABS(x)`

Returns the absolute value of a floating-point number that is, without any `+/-` sign for example `ABS(-10.099)` is `10.099`

If `x` is an integer, you won’t get an error, but the result will be converted to floating-point for example `ABS(-6)` is `6.0`. Use IABS to return the absolute value as a long integer.

ACOS

Usage: `a=ACOS(x)`

Returns the arc cosine, or inverse cosine (COS^{-1}) of `x`.

`x` must be in the range -1 to +1. The number returned will be an angle in radians. To convert the angle to degrees, use the DEG function.

ADDR

Usage: `a&=ADDR(`*`variable`*`)`

❸ `a%=ADDR(`*`variable`*`)`

Returns the address at which *`variable`* is stored in memory.

The values of different types of variables are stored in bytes starting at `ADDR(`*`variable`*`)`. See PEEK for details.

The maximum address is guaranteed to be less than 64K on the Series 3c, while it is not on the Series 5. The return type therefore must be a long integer on the Series 5, but may be an integer on the Series 3c.

❺ See SETFLAGS if you require the 64K limit to be enforced on the Series 5. If the flag is set to restrict the limit, `a&` is guaranteed to fit into an integer.

See UADD, USUB.

ADJUSTALLOC

Usage: ❺ `pcelln&=ADJUSTALLOC(pcell&,off&,am&)`

❸ `pcelln%=ADJUSTALLOC(pcell%,off%,am%)`

Opens or closes a gap at `off&` (`off%`) within the allocated cell `pcell&` (`pcell%)`, returning the new cell address or zero if out of memory. `off&` (`off%)` is 0 for the first byte in the cell. Opens a gap if the amount `am&` `(am%)` is positive, and closes it if negative.

The number of bytes allocated is restricted to 64K on the Series 3c, while it is not on the Series 5. The return type therefore must be a long integer on the Series 5, but may be an integer on the Series 3c.

❺ Cells are allocated lengths that are the smallest multiple of four greater than the size requested. An error will be raised if the cell address argument is not in the range known by the heap.

See also SETFLAGS if you require the 64K limit to be enforced on the Series 5. If the flag is set to restrict the limit, `pcelln&` is guaranteed to fit into an integer.

See ALLOC. See also the ‘Advanced.pdf’ document.

ALERT

Usage: any of

```
r%=ALERT(m1$,m2$,b1$,b2$,b3$)
r%=ALERT(m1$,m2$,b1$,b2$)
r%=ALERT(m1$,m2$,b1$)
r%=ALERT(m1$,m2$)
r%=ALERT(m1$)
```

Presents an alert - a simple dialog - with the messages and keys specified, and waits for a response. `m1$` is the message to be displayed on the first line, and `m2$` on the second line. If `m2$` is not supplied or if it is a null string, the second message line is left blank.

Up to three keys may be used. `b1$`, `b2$` and `b3$` are the strings (usually words) to use over the keys. `b1$` appears over an Esc key, `b2$` over Enter, and `b3$` over Space. This means you can have Esc, or Esc and Enter, or Esc, Enter and Space keys. If no key strings are supplied, the word `CONTINUE` is used above an Esc key.

The key number 1 for Esc, 2 for Enter or 3 for Space is returned.

❺ Constants for these return 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.

ALLOC

Usage: ❺ `pcell&=ALLOC(size&)`

❸ `pcell%=ALLOC(size%)`

Allocates a cell on the heap of the specified size, returning the pointer to the cell or zero if there is not enough memory.

The number of bytes allocated is restricted to 64K on the Series 3c, while it is not on the Series 5. The return type therefore must be a long integer on the Series 5, but may be an integer on the Series 3c.

❺ Cells are allocated lengths that are the smallest multiple of four greater than the size requested. An error will be raised if the cell address argument is not in the range known by the heap.

See also SETFLAGS if you require the 64K limit to be enforced. If the flag is set to restrict the limit, `pcelln&` is guaranteed to fit into an integer.

See ADJUSTALLOC, REALLOC, FREEALLOC. See also the ‘Advanced.pdf’ document.

APP

Usage: ❺

```
APP caption,uid&
...
ENDA
```

❸

```
APP caption
...
ENDA
```

Begins definition of an OPL application. *`caption`* is the application’s name (or caption) in the machine’s default language. Note that although *`caption`* is a string, it is not enclosed in quotes.

❺ `uid&` is the application’s UID. For distributed applications, official reserved UIDs must be used. These can be obtained by contacting Psion Software plc (see ‘OPL applications’ in the ‘Advanced.pdf’ document for details of how to do this).

All information included in the APP…ENDA structure will be used to generate an AIF file which specifies the applications caption in various languages, its icons for use on the System screen and its setting of FLAGS. See the ‘Advanced.pdf’ document for further details of this.

See CAPTION, ICON, FLAGS.

See also the ‘Advanced.pdf’ document for more details of OPAs.

APPEND

Usage: `APPEND`

Adds a new record to the end of the current data file. The record which was current is unaffected. The new record, the last in the file, becomes the current record.

The record added is made from the current values of the field variables `A.field1$`, `A.field2$`, and so on, of the current data file. If a field has not been assigned a value, zero will be assigned to it if it is a numeric field, or a null string if it is a string field.

Example:

```
PROC add:
  OPEN “address”,A,f1$,f2$,f3$
  PRINT “ADD NEW RECORD”
```

```
    PRINT “Enter name:”,
    INPUT A.f1$
    PRINT “Enter street:”,
    INPUT A.f2$
    PRINT “Enter town:”,
    INPUT A.f3$
    APPEND
    CLOSE
ENDP
```

To overwrite the current record with new field values, use UPDATE.

❺ **On the Series 5, INSERT, PUT and CANCEL should be used in preference to APPEND and UPDATE**, although APPEND and UPDATE are still supported for Series 3c compatibility. However, note that APPEND can generate a lot of extra (intermediate) erased records. COMPACT should be used to remove them, or alternatively use SETFLAGS to set auto-compaction on.

See the ‘Series 5 Database Handling’ section of the ‘Database.pdf’ document for more details. See also INSERT, MODIFY, PUT, CANCEL, SETFLAGS.

❸ APPENDSPRITE

Usage: `APPENDSPRITE time%,bit$(), dx%,dy%`

or `APPENDSPRITE time%,bit$()`

Appends a single bitmap-set to the current sprite.

`time%` gives the duration in tenths of seconds for the bitmap-set to be displayed before going on to the next bitmap-set in the sequence.

`bit$()` contains the names of bitmap files in the set, or `“”` to specify no bitmap. The array must have at least 6 elements:

`bit$(1)` for setting black pixels

`bit$(2)` for clearing black pixels

`bit$(3)` for inverting black pixels

`bit$(4)` for setting grey pixels

`bit$(5)` for clearing grey pixels

`bit$(6)` for inverting grey pixels

All the bitmaps in a single bitmap-set must be the same size or ‘Invalid argument’ error (-2) is raised on attempting to draw the sprite. Bitmaps in different bitmap-sets may differ in size.

`dx%` and `dy%`, if supplied, are the (x,y) offsets from the sprite position to the top-left of this bitmap-set, with positive for right and down. The default value of each is zero.

❺ On the Series 5, sprites are handled by the built-in Sprite OPX. See the ‘OPX.pdf’ document for more details.

ASC

Usage: a%=ASC(a$)

Returns the character code of the first character of a$.

For the Series 5, see Appendix D for the character set and for the Series 3c, see the character set in the back of the User Guide for the character codes. Alternatively, use A%=%char to find the code for char - e.g. %X for 'X'.

If a$ is a null string (“”) ASC returns the value 0.

Example: A%=ASC(“hello”) returns 104, the code for h.

ASIN

Usage: a=ASIN(x)

Returns the arc sine, or inverse sine (SIN^{-1}) of x.

x must be in the range -1 to +1. The number returned will be an angle in radians. To convert the angle to degrees, use the DEG function.

AT

Usage: AT x%,y%

Positions the cursor at x% characters across the text window and y% rows down. AT 1,1 always moves to the top left corner of the window. Initially, the window is the full size of the screen, but you can change its size and position with the SCREEN command.

A common use of AT is to display strings at particular positions in the text window. For example:

```
AT 5,2 :PRINT “message”.
```

- PRINT statements without an AT display at the left edge of the window on the line below the last PRINT statement (unless you use , or ;) and strings displayed at the top of the window eventually scroll off as more strings are displayed at the bottom of the window.
- Displayed strings always overwrite anything that is on the screen - they do not cause things below them on the screen to scroll down.

Example:

```
PROC records:
   LOCAL k%
   OPEN “clients”,A,name$,tel$
   DO
      CLS
      AT 1,7
      PRINT “Press a key to”
      PRINT “step to next record”
      PRINT “or Q to quit”
      AT 2,3 :PRINT A.name$
      AT 2,4 :PRINT A.tel$
      NEXT
```

```
    IF EOF
        AT 1,6 :PRINT “EndOfFile”
        FIRST
    ENDIF
    k%=GET
  UNTIL k%=%Q OR k%=%q
  CLOSE
ENDP
```

ATAN

Usage: `a=ATAN(x)`

Returns the arc tangent, or inverse tangent (TAN^{-1}) of `x`.

The number returned will be an angle in radians. To convert the angle to degrees, use the DEG function.

BACK

Usage: `BACK`

Makes the previous record in the current data file the current record.

If the current record is the first record in the file, then the current record does not change.

BEEP

Usage: `BEEP time%,pitch%`

Sounds the buzzer. The beep lasts for `time%`/32 seconds so for a beep a second long make `time%`=32, etc. The maximum is 3840 (2 minutes).

The pitch (frequency) of the beep is 512/(`pitch%`+1) KHz.

`BEEP 5,300` gives a comfortably pitched beep.

If you make `time%` negative, BEEP first checks whether the sound system is in use (perhaps by another OPL program), and returns if it is. Otherwise, BEEP waits until the sound system is free.

Example a scale from middle C:

```
PROC scale:
  LOCAL freq,n%                 REM n% relative to middle A
  n%=3                          REM start at middle C
  WHILE n%<16
  freq=440*2**(n%/12.0)         REM middle A = freq 440Hz
  BEEP 8,512000/freq-1.0
  n%=n%+1
  IF n%=4 OR n%=6 OR n%=9 OR n%=11 OR n%=13
    n%=n%+1
  ENDIF
  ENDWH
ENDP
```

❸ Alternatively, sound the buzzer with this statement: `PRINT CHR$(7)`. This beeps at a fixed pitch for a fixed length of time.

❺ Alternatively, sound the buzzer with this statement: `PRINT CHR$(7)`. This produces a click sound.

⚠ Note that on the Series 5 if your batteries are low BEEP may not produce the desired effect: the buzzer will be used instead to produce the ‘beep’. The buzzer produces a higher pitched sound.

❺ BEGINTRANS

Usage: `BEGINTRANS`

Begins a transaction on the current database. The purpose of this is to allow changes to a database to be committed in stages. Once a transaction has been started on a view (or table) then all database keywords will function as usual, but the changes to that view will not be made until COMMITTRANS is used.

See also COMMITTRANS, ROLLBACK, INTRANS.

❺ BOOKMARK

Usage: `b%=BOOKMARK`

Puts a bookmark at the current record of the current database view. The value returned can be used in GOTOMARK to make the record current again. Use KILLMARK to delete the bookmark.

BREAK

Usage: `BREAK`

Makes a program performing a DO...UNTIL or WHILE...ENDWH loop exit the loop and immediately execute the line following the UNTIL or ENDWH statement.

Example:

```
DO
  ...
  IF a=b
    BREAK
  ENDIF
  ...
UNTIL a=b
x%=3
```

BUSY

Usage: any of

```
BUSY str$,c%,delay%
BUSY str$,c%
BUSY str$
BUSY OFF
```

`BUSY str$` displays `str$` in the bottom left of the screen, until `BUSY OFF` is called. Use this to indicate ‘Busy’ messages, usually when an OPL program is going to be unresponsive to keypresses for a while.

If `c%` is given, it controls the corner in which the message appears:

`c%`	*corner*
0	top left
1	bottom left (default)
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.

`delay%` specifies 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.

Only one message can be shown at a time. The maximum string length of a BUSY message is 80 characters on the Series 5, and an ‘Invalid argument’ error is returned for any value in excess of this. On the Series 3c, the maximum length is 19 characters.

❺ BYREF

Usage: `BYREF` *`variable`*

Passes variable by reference to an OPX procedure when used in a procedure argument list. This means that the value of the variable may be changed by the procedure.

See the ‘OPX.pdf’ document for more details.

❸ CACHE

Usage: any of

```
CACHE init%,max%
CACHE ON
CACHE OFF
```

CACHE creates a procedure cache of a specified initial number of bytes `init%` which may grow up to the maximum size `max%`. You should usually TRAP this.

Once a cache has been created, `CACHE OFF` prevents further cacheing, although the cache is still searched when calling subsequent procedures. `CACHE ON` may then be used to re-enable cacheing.

❺ Series 5 procedures are automatically cached, so this command is not required.

❸ CACHEHDR

Usage: `CACHEHDR addr(hdr%())`

Read the current cache index header into array `hdr%()`, which must have at least 11 integer elements.

See the ‘Advanced.pdf’ document for more details.

❺ Series 5 procedures are automatically cached, so this command is not required.

❸ CACHEREC

Usage: `CACHEREC addr(rec%()),off%`

Read the cache index record at offset `off%` into array `rec%()`, which must have at least 18 integer elements.

See the ‘Advanced.pdf’ document for more details.

❺ Series 5 procedures are automatically cached, so this command is not required.

❸ CACHETIDY

Usage: CACHETIDY

Remove from the cache any procedures that have returned to their callers.

❺ Series 5 procedures are automatically cached, so this command is not required.

❸ CALL

Usage: `e%=CALL(s%,bx%,cx%,dx%,si%,di%)`

This function enables you to make Operating System calls. To use it requires **extensive** knowledge of the Operating System and related programming techniques. The syntax of this command is included here for completeness only.

The INT number itself is the least significant byte of `s%`. The AH value (the subfunction number) is the most significant byte of `s%`. The values of the other arguments are passed to the corresponding 8086 registers. The value of the AX register is returned.

❺ The Series 5 supports calls to the operating system using OPXs. Full description of their design is beyond the scope of this manual and is documented instead in the EPOC32 C++ Software Development Kit (SDK) which is available from Psion Software plc. See the ‘OPX.pdf’ document for details of built-in OPXs.

❺ CANCEL

Usage: `CANCEL`

Marks the end of a database’s INSERT or MODIFY phase and discards the changes made during that phase.

❺ CAPTION

Usage: `CAPTION caption$,languageCode%`

Specifies an OPA’s *public name* (or *caption*) for a particular language, which will appear below its icon on the Extras bar and in the list of ‘Programs’ in the ‘New File’ dialog (assuming the setting of FLAGS allows these) when the language is that used by the machine. CAPTION may only be used inside an APP...ENDA construct.

The language code specifies for which language variant the caption should be used, so that the caption need not be changed when used on a different language machine. If used, for whatever language, CAPTION causes the default caption given in the APP declaration to be discarded. Therefore CAPTION statements must be supplied for **every** language in which the application is liable to be used, including the language of the machine on which the application is originally developed.

The values of the language code should be one of the following:

English 1	French 2	German 3	Spanish 4
Italian 5	Swedish 6	Danish 7	Norwegian 8
Finnish 9	American 10	Swiss-French 11	Swiss-German 12
Portuguese 13	Turkish 14	Icelandic 15	Russian 16
Hungarian 17	Dutch 18	Belgian-Flemish 19	Australian 20
Belgian-French 21	Austrian 22	New Zealand 23	International French 24

Constants for the language codes 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.

The maximum length of `caption$` is 255 characters. However, you should bear in mind that a caption longer than around 8 characters will not fit neatly below the application’s icon on the Extras bar.

See APP. See also ‘OPL applications’ in the ‘Advanced.pdf’ document.

❸ CHANGESPRITE

Usage: `CHANGESPRITE ix%,time%,bit$(),dx%,dy%`

or `CHANGESPRITE ix%,time%,bit$()`

Changes the bitmap-set specified by `ix%` (1 for the first bitmap-set) in the current sprite, using the supplied bitmap files, offsets and duration in the same way as for APPENDSPRITE.

❺ On the Series 5, sprites are handled by a built-in OPX. See the ‘OPX.pdf’ document for more details.

CHR$

Usage: `a$=CHR$(x%)`

Returns the character with character code `x%`.

You can use it to display characters not easily available from the keyboard. For example, the instruction `PRINT CHR$(133)` displays an ellipsis (…).

The full character set for the Series 5 is in Appendix D in the ‘Appends.pdf’ document. For the Series 3c, see the User Guide.

❺ CLEARFLAGS

Usage: `CLEARFLAGS flags&`

Clears the flags given in `flags&` if they have previously been set by SETFLAGS, returning to the default.

See SETFLAGS.

CLOSE

Usage: `CLOSE`

❺ Closes the current view on a database. If there are no other views open on the database then the database itself will be closed. See SETFLAGS for details of how to set auto-compaction on closing files.

❸ Closes the current file (that is, the one which has been OPENed and most recently USEd).

If you've used ERASE to remove some records, CLOSE recovers the memory used by the deleted records, provided it is held either in the internal memory, on a memory disk (on the Series 5) or on a RAM SSD (on the Series 3c).

❸ CLOSESPRITE

Usage: `CLOSESPRITE id%`

Closes the sprite with ID `id%`.

❺ On the Series 5, sprites are handled by a built-in OPX. See the 'OPX.pdf' document for more details.

CLS

Usage: `CLS`

Clears the contents of the text window.

The cursor then goes to the beginning of the top line. If you have used CURSOR OFF the cursor is still positioned there, but is not displayed.

CMD$

Usage: `c$=CMD$(x%)`

Returns the command-line arguments passed when starting a program. Null strings may be returned. `x%` should be from 1 to 3 for the Series 5 and may be up to 5 on the Series 3c. `CMD$(2)` to `CMD$(5)` are only for OPL applications.

`CMD$(1)` returns the full path name used to start the running program.

`CMD$(2)` returns the full path name of the file to be used by an OPL application. On the Series 5, if the `CMD$(3)`=“`R`” (see below), a default filename, including path, is passed in `CMD$(2)`.

`CMD$(3)` returns “`C`” for “Create file” or “`O`” for “Open file” and may also return “`R`” on the Series 5. If the OPL application is being run with a new filename, this will return “`C`”. This happens the very first time the OPL application is used, and whenever a new filename is used to run it. Otherwise, the OPA is being run with the name of an existing file, and `CMD$(3)` will return “`O`” if it is selected directly from the system screen. “`R`” (Series 5 only) is returned if your application has been run from the Program editor or has been selected via the Application's icon on the Extras bar, and not by the selection or creation of one of your documents from the system screen.

❺ Constants for the array elements (1-3) and for the return values of `CMD$(3)` 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.

❸ `CMD$(4)` returns the *alias information*, if any. In practice this has no relevance for OPL applications.

❸ `CMD$(5)` returns the application name, as declared with the APP keyword.

See the ‘Advanced.pdf’ document for more details of OPL applications.

See also GETCMD$.

❺ COMMITTRANS

Usage: `COMMITTRANS`

Commits the transaction on the current view.

See also BEGINTRANS, ROLLBACK, INTRANS.

❺ COMPACT

Usage: `COMPACT file$`

Compacts the database `file$`, rewriting the file in place. All views on the database and the hence the file itself should be closed before calling this command. This should not be done to often since it uses considerable processor power.

Compaction can also be done automatically on closing a file by setting the appropriate flag using SETFLAGS.

❸ COMPRESS

Usage: `COMPRESS src$,dest$`

Copies data file `src$` to another data file `dest$`. If `dest$` already exists, the records in `src$` are appended to the end of `dest$`.

Deleted records are not copied. This makes COMPRESS particularly useful when copying from a Flash SSD. (The space used by deleted records on a RAM SSD or in internal memory is automatically freed when you close the file.)

If you want `src$` to overwrite instead of append to `dest$`, use, `TRAP DELETE dest$` before the COMPRESS statement.

You can use wildcards if you wish to copy more than one file at a time, but if the first name contains any wildcards, the second name must not include a filename, just the device and directory to which the files are to be copied under their original names.

Example: to copy all the data files on A: (in `\OPD`, the default directory) to `B:\BCK\`:

```
COMPRESS “A:*.ODB”,”B:\BCK\”
```

(Remember the final backslash on the directory name.)

See COPY for copying any type of file.

❺ This command is replaced by COMPACT on the Series 5.

CONTINUE

Usage: `CONTINUE`

Makes a program immediately go to the UNTIL... line of a DO...UNTIL loop or the WHILE... line of a WHILE...ENDWH loop i.e. to the test condition.

Example:

```
DO
  ...
  IF a<3.5
    CONTINUE
  ENDIF
  ...
UNTIL a=b
...
```

See also BREAK.

❺ CONST

Usage: `CONST` *`KConstantName=constantValue`*

Declares constants which are treated as literals, not stored as data. The declarations must be made outside any procedure, usually at the beginning of the module. *`KConstantName`* has the normal type-specification indicators (`%`, `&`, `$` or nothing for floating-point numbers). CONST values have global scope, and are not overridden by locals or globals with the same name: in fact the translator will not allow the declaration of locals or globals of the same name. By convention, all constants should be named with a leading `K` to distinguish them from variables.

It should be noted that it is not possible to define constants with values -32768 (for integers) and -214748648 (for long integers) in decimals, but hexadecimal notation may be used instead (i.e. values of $8000 and &80000000 respectively).

COPY

Usage: `COPY src$,dest$`

Copies the file `src$`, which may be of any type, to the file `dest$`. Any existing file with the name `dest$` is deleted. You can copy across devices. On the Series 3c you can use the appropriate file extensions to indicate the type of file, and on all machines use wildcards if you wish to copy more than one file at a time.

❺ If `src$` contains wildcards, `dest$` may specify either a filename similarly containing wildcards or just the device and directory to which the files are to be copied under their original names.

Example: To copy all the files from internal memory (in `\OPL`) to `D:\ME\`:

```
COPY "C:\OPL\*","D:\ME\"
```

(Remember the final backslash on the directory name.)

❸ If `src$` contains wildcards, `dest$` must not specify a filename, just the device and directory to which the files are to be copied under their original names.

You must specify either an extension or .* on the first filename. The file type extensions are listed in the User Guide.

Example: To copy all the OPL files from internal memory (in `\OPL`) to `B:\ME\`:

```
COPY “M:\OPL\*.OPL”,”B:\ME\”
```

(Remember the final backslash on the directory name.)

See COMPRESS for more control over copying data files. If you use COPY to copy a data file, deleted records are copied and you cannot append to another data file.

There are more details of full file specifications in the ‘Advanced.pdf’ document.

COS

Usage: `c=COS(x)`

Returns the cosine of `x`, where `x` is an angle in radians.

To convert from degrees to radians, use the RAD function.

COUNT

Usage: `c%=COUNT`

Returns the number of records in the current data file.
This number will be 0 if the file is empty.

❺ If you try to count the number of records in a view while updating the view an ‘Incompatible update mode’ error will be raised (This will occur between assignment and APPEND / UPDATE or between MODIFY / INSERT and PUT).

CREATE

❺ Usage: `CREATE tableSpec$,log,f1,f2,...`

Creates a table in a database. The database is also created if necessary. Immediately after calling CREATE, the file and view (or table) is open and ready for access.

`tableSpec$` contains the database filename and optionally a table name and the field names to be created within that table. For example:

```
CREATE “clients FIELDS name(40), tel TO phone”, D, n$, t$
```

The filename is `clients`. The table to be created within the file is `phone`. The comma-separated list, between the keywords `FIELDS` and `TO`, specifies the field names whose types are specified by the field handles (i.e. `n$`, `t$`).

The `name` field has a length of 40 bytes, as specified within the brackets that follow it. The `tel` field has the default length of 255 bytes. This mechanism is necessary for creating some indexes. See the ‘OPX.pdf’ document for more details on index creation.

- The filename may be a full file specification of up to 255 characters. A field name may be up to a maximum of 64 characters long. Text fields have a default length of 255 bytes.
- `log` specifies the logical file name A to Z. This is used as an abbreviation for the file name when you use other data file commands such as USE.

COMPATIBILITY WITH THE SERIES 3C

As on the Series 3c, the table specification may contain just the filename. In this case the table name will default to `Table1` and the field names will be derived from the handles: “$” replaced by “s”, “%” by “i”, and “&” by “a”. E.g. `n$` becomes `ns`. Knowing this allow views to be opened on tables (called `Table1`) that were created with the OPL16 method. However, it would be better to create the fields with proper names in the first place.

For example:

```
CREATE “clients”,A,n$,t%,d&
```

is a short version of

```
CREATE “clients FIELDS ns,ti,da TO Table1”,A,n$,t%,d&
```

both creating `Table1`. Database `clients` is also created if it does not yet exist.

❸ Usage: `CREATE file$,log,f1,f2,…`

Creates a data file called `file$`.

- The filename may be a full file specification of up to 128 characters. Field names may be up to 8 letters/numbers.
- The file may have up to 32 fields, as specified by `f1`, `f2`... (if viewed in the Data application, field `f1` starts on the top line of the window, `f2` is below it, etc.)
- The logical view name `log` can be any letter in the range `A` to `D` and is used to identify the view to other database commands such as USE.

Immediately after the CREATE statement, the file is open and can be accessed.

Example:

```
CREATE “CLIENTS”,B,NM$,PHON$
```

would create a data file in the internal memory with the name `CLIENTS` and the logical name `B`.

See also the ‘Data File Handling’ and ‘Series 5 Database Handling’ sections of the ‘Database.pdf’ document.

❸ CREATESPRITE

Usage: `id%=CREATESPRITE`

Creates a sprite, returning the sprite ID.

❺ On the Series 5, sprites are handled by a built-in OPX. See the ‘OPX.pdf’ document for more details.

CURSOR

Usage: any of

```
CURSOR ON
CURSOR OFF
CURSOR id%,asc%,w%,h%
CURSOR id%,asc%,w%,h%,type%
CURSOR id%
```

`CURSOR ON` switches the text cursor on at the current cursor position. Initially, no cursor is displayed.

You can switch on a graphics cursor in a window by following CURSOR with the ID of the window. This replaces any text cursor. At the same time, you can also specify the cursor's shape, and its position relative to the baseline of text.

`asc%` is the *ascent* - the number of pixels (-128 to 127) by which the top of the cursor should be above the baseline of the current font. `h%` and `w%` (both from 0 to 255) are the cursor's height and width.

If you do not specify them, the following default values are used:

`asc%`	font ascent
`h%`	font height
`w%`	2

If `type%` is given, it can have these effects:

2	not flashing
4	grey

You can add these values together to combine effects - if `type%` is 6 a grey non-flashing cursor is drawn. The Series 3c also supports an obloid cursor by specifying a `type%` of 1. Using `type%=1` on the Series 5 just displays a default graphics cursor, as though no type had been specified.

❺ Constants for these types 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.

An error is raised if `id%` specifies a bitmap rather than a window.

`CURSOR OFF` switches off any cursor.

DATETOSECS

Usage: `s&=DATETOSECS(yr%,mo%,dy%,hr%,mn%,sc%)`

Returns the number of seconds since 00:00 on 1/1/1970 at the date/time specified.

Raises an error for dates before 1/1/1970.

The value returned is an **unsigned** long integer. (Values up to +2147483647, which is 03:14:07 on 19/1/2038, are returned as expected. Those from +2147483648 upwards are returned as negative numbers, starting from -2147483648 and increasing towards zero.)

See also SECSTODATE, HOUR, MINUTE, SECOND.

DATIM$

Usage: `d$=DATIM$`

Returns the current date and time from the system clock as a string - for example: `“Fri 16 Oct 1992 16:25:30”`. The string returned always has this format - 3 mixed-case characters for the day, then a space, then 2 digits for the day of the month, and so on.

❺ Constants for offsets of each elements within the string (to be used with MID$, for example) 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. The Date OPX provides a large set of procedures for manipulating dates and for accurate timing. See the ‘OPX.pdf’ document for more details.

DAY

Usage: `d%=DAY`

Returns the current day of the month (1 to 31) from the system clock.

DAYNAME$

Usage: `d$=DAYNAME$(x%)`

Converts `x%`, a number from 1 to 7, to the day of the week, expressed as a three letter string. E.g. `d$=DAYNAME$(1)` returns `MON`.

Example:

```
PROC Birthday:
  LOCAL d&,m&,y&,dWk%
  DO
    dINIT
    dTEXT “”,”Date of birth”,2
    dTEXT “”,”eg 23 12 1963”,$202
    dLONG d&,”Day”,1,31
    dLONG m&,”Month”,1,12
    dLONG y&,”Year”,1900,2155
    IF DIALOG=0 :BREAK :ENDIF
    dWk%=DOW(d&,m&,y&)
    CLS :PRINT DAYNAME$(dWk%),
    PRINT d&,m&,y&
    dINIT dTEXT “”,”Again?”,$202
    dBUTTONS “No”,%N,”Yes”,%Y
  UNTIL DIALOG<>%y
ENDP
```

See also DOW.

DAYS

Usage: `d&=DAYS(day%,month%,year%)`

Returns the number of days since 1/1/1900.

Use this to find out the number of days between two dates.

Example:

```
PROC deadline:
   LOCAL a%,b%,c%,deadlin&
   LOCAL today&,togo%
   PRINT “What day? (1-31)”
   INPUT a%
   PRINT “What month? (1-12)”
   INPUT b%
   PRINT “What year? (19??)”
   INPUT c%
   deadlin&=DAYS(a%,b%,1900+c%)
   today&=DAYS(DAY,MONTH,YEAR)
   togo%=deadlin&-today&
   PRINT togo%,”days to go”
   GET
ENDP
```

See also dDATE, SECSTODATE.

❺ The Date OPX provides a large set of procedures for manipulating dates and for accurate timing. See the ‘OPX.pdf’ document for more details.

❺ DAYSTODATE

Usage: `DAYSTODATE days&,year%,month%,day%`

This converts `days&`, the number of days since 1/1/1900, to the corresponding date, returning the day of the month to `day%`, the month to `month%` and the year to `year%`. This is useful for converting the value set by dDATE, which also gives days since 1/1/1900.

DBUTTONS

Usage: any of

```
dBUTTONS p1$,k1%,p2$,k2%,p3$,k3%
```

```
dBUTTONS p1$,k1%,p2$,k2%
```

```
dBUTTONS p1$,k1%
```

❺ The Series 5 allows more than 3 buttons which may be added in the same way.

Defines exit keys to go at the bottom (or the side on the Series 5: see dINIT) of a dialog.

From one to three (or more on the Series 5) exit keys may be defined. Each pair of `p$` and `k%` specifies an exit key; `p$` is the text to be displayed on it (above it on the Series 3c), while `k%` is the keycode of the shortcut key. DIALOG returns the keycode of the key pressed (in lower case for letters).

For alphabetic keys, use the `%` sign - `%A` means ‘the code of `A`’, and so on. The shortcut key is then Ctrl+alphabetic key on the Series 5, but just the alphabetic key without a modifier on the Series 3c. An appendix lists the codes for keys (such as Tab) which are not part of the character set. If you use the code for one of these keys, its name (e.g. ‘Tab’, or ‘Enter’) will be shown in the key.

❺ On the Series 5, the following effects may be obtained by adding the appropriate constants to the shortcut key keycode:

effect	*constant value*
display a button with no shortcut key label underneath it	256 ($100)
use the key alone (without the Ctrl modification) as the shortcut key	512 ($200)

Constants for these flags and keycodes for Esc and other keys 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.

If you use a negative value for a `k%` argument, that key is a ‘Cancel’ key. The corresponding positive value is used for the key to display and the value for DIALOG to return, but if you do press this key to exit, the *`var`* variables used in the commands like dEDIT, dTIME etc. will not be set. For the Series 5, when using a negative shortcut to specify the cancel button, you must negate the shortcut together with any added flags.

The Esc key will always cancel a dialog box, with DIALOG returning 0. If you want to show the Esc key as one of the exit keys, use -27 as the `k%` argument (its keycode is 27) so that the var variables will **not** be set if Esc is pressed.

There can be only one dBUTTONS item per dialog.

❺ The buttons take up two lines on the screen. dBUTTONS may be used anywhere between dINIT and DIALOG; the postion of its use does not affect the position of the buttons in the dialog.

❸ The buttons take up three lines on the screen. dBUTTONS must be the last dialog command you use before DIALOG itself.

Some keypresses cannot be specified, for example, those using the Control key for the Series 3c.

This example presents a simple query, returning ‘False’ for No, or ‘True’ for Yes, providing shortcut keys of `N` and `Y` respectively and without labels beneath the keys.

```
PROC query:
  dINIT
  dTEXT “”,“FORGET CHANGES”,2
  dTEXT “”,“Sure?”,$202
  dBUTTONS “No”,-(%N OR $100 OR $200),“Yes”,%Y OR $100 OR $200
  RETURN DIALOG=%y
ENDP
```

❸ On the Series 3c, the same shortcut keys can be specified using, although labels are always visible on the Series 3c:

```
dBUTTONS “No”,%N,“Yes”,%Y
```

See also dINIT.

❺ DCHECKBOX

Usage: `dCHECKBOX chk%,prompt$`

Creates a dialog *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.

See also dINIT.

DCHOICE

Usage:

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

or ❺

```
dCHOICE var choice%,p$,list1$+”,...”
dCHOICE var choice%,””,list2$+”,...”
...
dCHOICE var choice%,””,listN$
```

Defines a choice list to go in a dialog.

`p$` will be displayed on the left side of the line. `list$` should contain the possible choices, separated by commas - for example, `“No,Yes”`. One of these will be displayed on the right side of the line, and ◀ ▶ can be used to move between the choices.

`choice%` must be a LOCAL or a GLOBAL variable. It specifies which choice should initially be shown — 1 for the first choice, 2 for the second, and so on. When you finish using the dialog, `choice%` is given a value indicating which choice was selected — again, 1 for the first choice, and so on.

❺ On the Series 5, dCHOICE supports an unrestricted number of items (up to memory limits). To extend a dCHOICE list, add a comma after the last item on the line followed by `“...”` (three full-stops), as shown in the usage above. `choice%` must be the same on all the lines, otherwise an error is raised. For example, the following specifies items i1, i2, i3, i4, i5, i6:

```
dCHOICE ch%,prompt$,”i1,i2,...”
dCHOICE ch%,””,”i3,14,...”
dCHOICE ch%,””,”i5,i6"
```

See also dINIT.

DDATE

Usage: `dDATE var lg&,p$,min&,max&`

Defines an edit box for a date, to go in a dialog.

`p$` will be displayed on the left side of the line.

`lg&`, which must be a LOCAL or a GLOBAL variable, specifies the date to be shown initially. Although it will appear on the screen like a normal date, for example `15/03/92`, `lg&` must be specified as “days since 1/1/1900”.

`min&` and `max&` give the minimum and maximum values which are to be allowed. Again, these are in days since 1/1/1900. An error is raised if `min&` is higher than `max&`.

When you finish using the dialog, the date you entered is returned in `lg&`, in days since 1/1/1900.

The system setting determines whether years, months or days are displayed first.

See also DAYS, SECSTODATE,DAYSTODATE, dINIT.

❺ DECLARE EXTERNAL

Usage: `DECLARE EXTERNAL`

Causes the translator to report an error if any variables or procedures are used before they are declared. It should be used at the beginning of the module to which it applies, before the first procedure. It is useful for detecting 'Undefined externals' errors at translate-time rather than at runtime.

For example, with DECLARE EXTERNAL commented out, the following translates and raises the error, 'Undefined externals, i' at runtime. Adding the declaration causes the error to be detected at translate-time instead.

```
REM DECLARE EXTERNAL
PROC main:
   LOCAL i%
   i%=10
   PRINT i
   GET
ENDP
```

If you use this declaration, you will need to declare all subsequent variables and procedures used in the module, using EXTERNAL.

See also EXTERNAL.

❺ DECLARE OPX

Usage:
```
DECLARE OPX opxname,opxUid&,opxVersion&
...
END DECLARE
```

Declares an OPX. *`opxname`* is the name of the OPX, `opxUid&` its UID and `opxVersion&` its version number.

Declarations of the OPX's preocedures should be made inside this structure.

See the 'OPX.pdf' document for more details.

DEDIT

Usage: `dEDIT var str$,p$,len%`

or `dEDIT var str$,p$`

Defines a string edit box, to go in a dialog.

`p$` will be displayed on the left side of the line.

`str$` is the string variable to edit. Its initial contents will appear in the dialog. The length used when `str$` was defined is the maximum length you can type in.

len%, if supplied, gives the width of the edit box (allowing for 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 possibly be.

See also dTEXT.

➎ DEDITMULTI

Usage: dEDITMULTI var ptrData&,p$,widthInChars%,numberLines%,maxLength%

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). It is also possible to paste text into the buffer from other applications and vice versa, although any formatting or embedded objects contained in text pasted in will be removed.

ptrData& is the address of a buffer to take the edited data. It could be the address of an array as returned by ADDR, or of a heap cell, as returned by ALLOC (see ADDR and ALLOC). The buffer may not be specified directly as a string and may not be read as such. Instead it should be peeked, byte by byte (see PEEK). The leading 4 bytes at ptrData& contain the initial number of bytes of data following. These bytes are also set by dEDITMULTI to the actual number of bytes edited. For this reason it is convenient to use a long integer array as the buffer, with at least 1+(maxLength%+3)/4 elements. The first element of the array then specifies the initial length.

If an allocated cell is used (probably because more than 64K is required), the first 4 bytes of the cell must be set to the initial length of the data. If this length is not set then an error will be raised. For example if a cell of 100000 bytes is allocated, you would need to poke a zero long integer in the start to specify that there is initially no text in the cell. For example:

```
p&=ALLOC(100000)
POKEL p&,0          REM Text starts at p&+4
```

Special characters such as line breaks and tab characters may appear in the buffer. Constants for the codes of these 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.

The prompt, p$ will be displayed on the left side of the edit box. widthInChars% specifies the width of the edit box within which the text is wrapped, using a notional average character width. The actual number of characters that will fit depends on the character widths, with e.g. more 'i's fitting than 'w's. numberLines% specifies the number of full lines displayed. Any more lines will be scrolled. maxLength% specifies the length in bytes of the buffer provided (excluding the bytes used to store the length).

The Enter key is used by a multi-line edit box which has the focus before being offered to any buttons. This means that Enter can't be used to exit the dialog, unless another item is provided that can take the focus without using the Enter key. Normal practice is to provide a button that does not use the Enter key to exit a dialog whenever it contains a multi-line edit box. The Esc key will always cancel a dialog however, even when it contains a multi-line edit box.

The following example presents a three-line edit box which is about 10 characters wide and allows up to 399 characters:

```
CONST KLenBuffer%=399
PROC dEditM:
  LOCAL buffer&(101)            REM 101=1+(399+3)/4 in integer arithmetic
  LOCAL pLen&,pText&
  LOCAL i%
  LOCAL c%
```

```
   pLen&=ADDR(buffer&(1))
   pText&=ADDR(buffer&(2))
   WHILE 1
     dINIT "Try dEditMulti"
     dEDITMULTI pLen&,"Prompt",10,3,KLenBuffer%
     dBUTTONS "Done",%d          REM button needed to exit dialog
     IF DIALOG=0 :BREAK :ENDIF
     PRINT "Length:";buffer&(1)
     PRINT "Text:"
     i%=0
     WHILE i%<buffer&(1)
          c%=PEEKB(pText&+i%)
          IF c%>=32
               PRINT CHR$(c%);
          ELSE
               PRINT ".";    REM just print a dot for special characters
          ENDIF
          i%=i%+1
     ENDWH
   ENDWH
ENDP
```

See also dINIT.

DEFAULTWIN

Usage: `DEFAULTWIN mode%`

Change the default window (ID=1) to enable or disable the use of grey (on the Series 3c) or change the colour mode (on the Series 5).

❺ **For the Series 5:**

The default window uses 4-colour mode initially.

`mode%=1` just clears the screen, leaving the window in 4-colour mode. Clearing of the screen ensures compatibility with Series 3c (see above). `mode%` of `0` changes the screen to 2-colour mode (actually results in a mapping of greys to white or black) and `mode%` of 2 changes to 16-colour mode, as expected.

Using DEFAULTWIN with either of these values also clears the screen.

Using 4-colour mode uses more power than using 2-colour mode and using 16-colour mode uses more power that either of these. See the ‘Graphics’ section for more information.

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.

❸ **For the Series 3c:**

Initially grey cannot be used in the default window.

`mode%=1` enables the use of grey. `mode%=0` disables the use of grey.

A side-effect of DEFAULTWIN is to clear the default window.

Using grey uses more memory than using black only.

You are advised to call DEFAULTWIN once and for all near the start of your program if you need to change the colour mode of the default window on the Series 5 or use grey on the Series 3c. If it fails with ‘Out of memory’ error, the program can then exit cleanly without losing vital information.

See also gGREY, gCOLOR, gCREATE.

DEG

Usage: `d=DEG(x)`

Converts from radians to degrees.

Returns `x`, an angle in radians, as a number of degrees. The formula used is: `180*x/PI`

All the trigonometric functions (SIN,COS etc.) work in radians, not degrees. You can use DEG to convert an angle returned by a trigonometric function back to degrees:

Example:

```
PROC xarctan:
  LOCAL arg,angle
  PRINT “Enter argument:”;
  INPUT arg
  PRINT “ARCTAN of”,arg,”is”
  angle=ATAN(arg)
  PRINT angle,”radians”
  PRINT DEG(angle),”degrees”
  GET
ENDP
```

To convert from degrees to radians, use RAD.

DELETE

Usage: `DELETE filename$`

Deletes any type of file.

❺ You can use wildcards for example, to delete all the files in `D:\OPL`

```
DELETE “D:\OPL\*”
```

❸ You can use wildcards for example, to delete all the OPL files in `B:\OPL`

```
DELETE “B:\OPL\*.OPL”
```

The file type extensions are listed in the User Guide.

See also RMDIR.

❺ DELETE

Usage: `DELETE dbase$,table$`

This deletes the table, `table$`, from the database, `dbase$`. To do this all views of the database, and hence the database itself, must be closed.

DFILE

Usage: `dFILE var file$,p$,f%`

or ❺ `dFILE var file$,p$,f%,uid1&,uid2&,uid3&`

Defines a filename edit box or selector, to go in a dialog. A ‘Folder’ and ‘Disk’ selector are automatically added on the following lines (on the Series 3c, a ‘Disk’ selector only).

❺ By default no prompts are displayed for the file, folder and disk selectors on the Series 5. A comma-separated prompt list should be supplied. For example, for a filename editor with the standard prompts use:

```
dFILE f$,”File,Folder,Disk”,1
```

❸ The disk selector is automatically supplied with a prompt on the Series 3c and `p$` will be the prompt on the left of the filename selector.

`f%` controls the type of file editor or selector, and the kind of input allowed. You can add together any of the following values:

	value	*meaning*
	0	use a selector
	1	use an edit box
	2	allow directory names
	4	directory names only
	8	disallow existing files
	16	query existing files
	32	allow null string input
❸	64	don’t display file extension
	128	obey/allow wildcards
❺	256	allow ROM files to be selected
❺	512	allow files in the System folder to be selected

The first of the list is the most crucial. If you add 1 into `f%`, you will see a file edit box, as when creating a new file. If you do not add 1, you will see the ‘matching file’ selector, used when choosing an existing file.

The value 64 (to omit file extensions) is not valid on the Series 5 since file extensions are no longer treated as special components of the filename.

If performing a ‘copy to’ operation, you might use 1+2+16, to specify a file edit box, in which you **can** type the name of a directory to copy to, and which will produce a query if you type the name of an existing file.

If asking for the name of a directory to remove, you might use 4, to allow an existing directory name only.

‘Query existing’ is ignored if ‘disallow existing’ is set. These two, as well as ‘allow null string input’, only work with file edit boxes, not ‘matching file’ selectors.

❺ For file selectors, dFILE supports file restriction by **UID**, or by **type** from the user’s point of view. Documents are identified by three UIDs which identify which application created the document and what kind of file it is. Specifying all three UIDs will restrict the files as much as is possible, and specifying fewer will provide less restriction. You can supply 0 for `uid1&` and `uid2&` if you only want to restrict the list to `uid3&`. This may be useful when dealing with documents from one of your own applications: you can easily find out the third UID as it will be the UID you specified in the APP statement. Note that UIDs are ignored for editors. For example, if your application has UID `KUidMyApp&`, then the following will list only your application-specific documents:

```
dFILE f$,p$,f%,0,KUidOplDoc&,KUidMyApp&
                              REM KUidOplDoc& for OPL docs
```

Some OPL-related UIDs are given in Const.oph. See the ‘Calling Procedures’ for details of how to use this file and Appendix E for a listing of it.

`file$` is the string variable to edit. Its initial contents always control the initial drive and directory used. Any filename part of file`$` is shown initially in the filename box. For a ‘matching file’ selector, you can use wildcards in the filename part (such as `*.tmp`) to control which filenames are matched. To do this, you must add 128 to `f%`. 128 also allows wildcard specifications to be **entered** (returned in `str$`), for both ‘matching’ and ‘new file’ selectors.

❸ On the Series 3c, if `str$` does not contain any drive or directory information, the path as set by SETPATH is used. If SETPATH has not been used, the `\OPD` directory on the default drive (usually `M:`, ‘Internal’) is used.

With a **matching** file selector (as opposed to an edit box) the value 8 restricts the selection to files which match the filename/extension in `file$`.

❸ Matching file selectors can also use 64, in which case files with the same extension as that in `file$` are shown without this extension. (Many Psion file selectors are like this.)

You can always press Tab to produce the full file selector with a dFILE item.

`file$` must be declared to be 255 bytes long, since file names may be up to this length, and if it is shorter an error will be raised. On the Series 3c, it must be at least 128 bytes long.

❺ Constants for the flags used by dFILE and for some OPL-related UIDs 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.

See also dINIT.

DFLOAT

Usage: `dFLOAT var fp,p$,min,max`

Defines an edit box for a floating-point number, to go in a dialog.

`p$` will be displayed on the left side of the line.

`min` and `max` give the minimum and maximum values which are to be allowed. An error is raised if `min` is higher than `max`.

`fp` must be a LOCAL or a GLOBAL variable. It specifies the value to be shown initially. When you finish using the dialog, the value you entered is returned in `fp`.

See also dINIT.

DIALOG

Usage: `n%=DIALOG`

Presents the dialog prepared by dINIT and commands such as dTEXT and dCHOICE. If you complete the dialog by pressing Enter, your settings are stored in the variables specified in dLONG, dCHOICE etc., although you can prevent this with dBUTTONS.

If you used dBUTTONS when preparing the dialog, the keycode which ended the dialog is returned. Otherwise, DIALOG returns the line number of the item which was current when Enter was pressed. The top item (or the title line, if present), has line number 1.

If you cancel the dialog by pressing Esc, the variables are not changed, and 0 is returned.

See also dINIT.

❸ DIAMINIT

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

Initialises the diamond list (discarding any existing list). `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%>=1` you must supply at least this many strings.**

If `pos%` is not supplied or if `pos%=0`, or if DIAMINIT is used on its own with no arguments, no bar is defined.

If `pos%=-1` the diamond bar is removed as for the small status window on the Series 3c.

❺ The Series 5 has no status windows. You should use a toolbar instead. See the ‘Friendlier Interaction’ section of the ‘GUI.pdf’ document for more details of toolbar usage.

❸ DIAMPOS

Usage: `DIAMPOS pos%`

Positions the diamond indicator on the diamond list.

Positioning outside the range of the items wraps around in the appropriate way. `pos%=0` causes the diamond symbol to disappear.

❺ The Series 5 has no status windows. You should use a toolbar instead. See the ‘Friendlier Interaction’ section of the ‘GUI.pdf’ document for more details of toolbar usage.

DINIT

Usage: any of

`dINIT`

`dINIT title$`

❺ `dINIT title$,flags%`

Prepares for definition of a dialog, cancelling any existing one. Use dTEXT, dCHOICE etc. to define each item in the dialog, then DIALOG to display the dialog.

If `title$` is supplied, it will be displayed at the top of the dialog.

❸ Any supplied `title$` will be centred and with a line across the dialog below it.

❺ Any supplied `title$` will be positioned in a grey box at the top of the dialog.

`flags%` can be any added combination of the following constants to achieve the following effects,

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

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.

On the Series 5, if an error occurs when adding an item to a dialog, the dialog is deleted and dINIT needs calling again. This is necessary to avoid having partially specified dialog lines.

In practical terms, this means that where the following artificial example would work on the Series 3c, giving just a long integer editor, it will raise a ‘Structure fault’ error on the Series 5.

```
     dINIT
     ONERR e1
     REM bad arg list gives argument error
     dCHOICE ch%,”ChList”,”a,b,,,,c”
e1::
     ONERR OFF
     dLONG l&,”Long”,0,12345
     DIALOG
```

DIR$

Usage: `d$=DIR$(filespec$)`

then `d$=DIR$(“”)`

Lists filenames, including subdirectory names, matching a file specification. You can include wildcards in the file specification. If `filespec$` is just a directory name, include the final backslash on the end for example, `“\TEMP\”` . Use the function like this:

- DIR$(filespec$) returns the name of the first file matching the file specification.
- DIR$(“”) then returns the name of the second file in the directory.
- DIR$(“”) again returns the third, and so on.

When there are no more matching files in the directory, `DIR$(“”)` returns a null string.

❺ Example, listing all the files whose names begin with `A` in `C:\ME\`

```
PROC dir:
  LOCAL d$(255)
  d$=DIR$(“C:\ME\A*”)
  WHILE d$<>“”
       PRINT d$
       d$=DIR$(“”)
  ENDWH
  GET
ENDP
```

❸ Example, listing all the `.DBF` files in `M:\DAT`:

```
PROC dir:
  LOCAL d$(128)
  d$=DIR$(“M:\DAT\*.DBF”)
  WHILE d$<>“”
       PRINT d$
       d$=DIR$(“”)
  ENDWH
  GET
ENDP
```

DLONG

Usage: `dLONG var lg&,p$,min&,max&`

Defines an edit box for a long integer, to go in a dialog.

`p$` will be displayed on the left side of the line.

`min&` and `max&` give the minimum and maximum values which are to be allowed. An error is raised if `min&` is higher than `max&`.

`lg&` must be a LOCAL or a GLOBAL variable. It specifies the value to be shown initially. When you finish using the dialog, the value you entered is returned in `lg&`.

See also dINIT.

DO...UNTIL

Usage:
```
DO
    statement
    ...
UNTIL condition
```

DO forces the set of statements which follow it to execute repeatedly until the `condition` specified by UNTIL is met.

This is the easiest way to repeat an operation a certain number of times.

Every DO must have its matching UNTIL to end the loop.

If you set a *`condition`* which is never met, the program will go round and round, locked in the loop forever.

❺ You can escape by pressing Ctrl+Esc, provided you haven't set `ESCAPE OFF`. If you have set `ESCAPE OFF`, you will have to return to go to the Task list, select your program in the list and click the 'Close file' option.

❸ You can escape by pressing Psion-Esc, provided you haven't set `ESCAPE OFF`. If you have set `ESCAPE OFF`, you will have to return to the System screen, move to the program name under the RunOpl icon, and press Delete.

DOW

Usage: `d%=DOW(day%,month%,year%)`

Returns the day of the week from 1 (Monday) to 7 (Sunday) given the date.

`day%` must be between 1 and 31, `month%` from 1 to 12 and `year%` from 1900 to 2155.

For example, `D%=DOW(4,7,1992)` returns 6, meaning Saturday.

❺ Constants for the numeric values assigned to the days of the week 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.

DPOSITION

Usage: `dPOSITION x%,y%`

Positions a dialog. 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.

❺ Constants for the positions 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.

See also dINIT.

❸ DRAWSPRITE

Usage: `DRAWSPRITE x%,y%`

Draws the current sprite in the current window with top-left at pixel position `x%,y%`.

❺ On the Series 5, sprites are handled by the built-in Sprite OPX. See the ‘OPX.pdf’ for more details.

DTEXT

Usage: `dTEXT p$,body$,t%`

or `dTEXT p$,body$`

Defines a line of text to be displayed in a dialog.

`p$` will be displayed on the left side of the line, and `body$` on the right side. If you only want to display a single string, use a null string (`""`) for `p$`, and pass the desired string in `body$`. It will then have the whole width of the dialog to itself. An error is raised if `body$` is a null string.

`body$` is normally displayed left aligned (although usually in the right column). You can override this by specifying `t%`:

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

However, note that on the Series 5, alignment of `body$` is only supported when `p$` is null, with the body being left aligned otherwise. In addition, you can add any or all of the following three values to `t%`, for these effects:

❸	$100	use bold text for `body$`
	$200	draw a line below this item
	$400	allow this item to be selected
❺	$800	specify this item as a text separator

Note that on the Series 5, bold dialog text is not supported. You can display a line separator between any dialog items by setting the flag $800 on an item which has null `p$` and `body$`. (If `p$` and/or `body$` are not null, then the flag is ignored and no separator is drawn.) The separator counts as an item in the value returned by DIALOG. On the Series 3c, only one line can be drawn across a dialog using the flag $200. It will be below the last item which asks for it, whether the title from dINIT (Series 3c only) or a dTEXT item. The flag $400 only allows the prompt, and not the body, to be selected.

❺ Constants for the text types 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.

See also dEDIT, dINIT

DTIME

Usage: `dTIME var lg&,p$,t%,min&,max&`

Defines an edit box for a time, to go in a dialog.

`p$` will be displayed on the left side of the line.

`lg&`, which must be a LOCAL or a GLOBAL variable, specifies the time to be shown initially. Although it will appear on the screen like a normal time, for example `18:27`, `lg&` must be specified as seconds after 00:00. A value of 60 means one minute past midnight; 3600 means one o'clock, and so on.

`min&` and `max&` give the minimum and maximum values which are to be allowed. Again, these are in seconds after 00:00. An error is raised if `min&` is higher than `max&`.

When you finish using the dialog, the time you entered is returned in `lg&`, in seconds after 00:00.

`t%` specifies the type of display required, as follows:

	`t%`	*time display*
	0	absolute time no seconds
	1	absolute time with seconds
	2	duration no seconds
	3	duration with seconds
❺	4	time without hours
❺	8	absolute time in 24 hour clock

❺ Constants for dTIME types 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, `03:45` represents an absolute time while 3 hours 45 minutes represents a duration.

Absolute times are displayed in 24-hour or am/pm format according to the current system setting. 8 displays the time in 24 hour clock, regardless of the system setting on the Series 5.

Absolute times always display am or pm as appropriate, unless 24 hour clock is being used. Durations never display am or pm. Note, however, that if you use the flag 4 (no hours) then the am/pm symbol **will** be displayed and the flag 2 must be added if you wish to hide it.

See also dINIT.

DXINPUT

Usage: `dXINPUT var str$,p$`

Defines a secret string edit box, such as for a password, to go in a dialog.

`p$` will be displayed on the left side of the line.

`str$` is the string variable to take the string you type.

⚠ `str$` must be less than 16 characters long on the Series 5 and must be at least eight characters long on the Series 3c.

Initially the dialog does not show any characters for the string; the initial contents of `str$` are ignored. A special symbol will be displayed for each character you type, to preserve the secrecy of the string.

See also dINIT.

EDIT

Usage: `EDIT a$`

Displays a string variable which you can edit directly on the screen. All the usual editing keys are available the arrow keys move along the line, Esc clears the line, and so on.

When you have finished editing, press Enter to confirm the changes. If you press Enter before you have made any changes, then the string will be unaltered.

If you use EDIT in conjunction with a PRINT statement, use a comma at the end of the PRINT statement, so that the string to be edited appears on the same line as the displayed string:

```
...
PRINT “Edit address:”,
EDIT A.address$
UPDATE
....
```

TRAP EDIT

If the Esc key is pressed while no text is on the input line, the ‘Escape key pressed’ error (number -114) will be returned by ERR provided that the EDIT has been trapped. You can use this feature to enable the user to press the Esc key to escape from inputting a string.

See also INPUT, dEDIT.

ELSE/ELSEIF/ENDIF

See IF.

ENDA

See APP.

ENDV

See VECTOR

ENDWH

See WHILE.

❸ ENTERSEND

Usage: `ret%=ENTERSEND(pobj%,m%,var p1,...)`

This is the same as SEND except that, if the method leaves, the error code is returned to the caller. Otherwise the value returned is as returned by the method.

❺ OPL now handles leaving without the need to use this function.

❸ ENTERSEND0

Usage: `ret%=ENTERSEND0(pobj%,m%,var p1,...)`

This is the same as ENTERSEND except that, if the method does **not** leave, zero is returned.

❺ OPL now handles leaving without the need to use this function.

EOF

Usage: `e%=EOF`

Finds out whether you're at the end of a file yet.

Returns -1 (true) if the end of the file has been reached, or 0 (false) if it hasn't.

When reading records from a file, you should test whether there are still records left to read, otherwise you may get an error.

Example:

```
PROC eoftest:
  OPEN “myfile”,A,a$,b%
  DO
    PRINT A.a$
    PRINT A.b%
    NEXT
    PAUSE -40
  UNTIL EOF
  PRINT “The last record”
  GET
  RETURN
ENDP
```

ERASE

Usage: `ERASE`

Erases the current record in the current file.

The next record is then current. If the erased record was the last record in a file, then following this command the current record will be null and EOF will return true.

ERR

Usage: `e%=ERR`

Returns the number of the last error which occurred, or 0 if there has been no error.

Example:

```
  ...
  PRINT “Enter age in years”
age::
  TRAP INPUT age%
  IF ERR=-1
    PRINT “Number please:”
    GOTO age
  ENDIF
    ...
```

❺ You can set the value returned by ERR to 0 (or any other value) by using `TRAP RAISE 0`. This is useful for clearing ERR.

❸ To clear the value of ERR on the Series 3c, you need to do the following,

```
  ONERR e0
  RAISE 0
e0::
  ONERR OFF
```

See also ERR$,ERRX$. See the ‘Errors.pdf’ document for full details, including the list of error numbers and messages.

ERR$

Usage: `e$=ERR$(x%)`

Returns the error message for the specified error code `x%`.

`ERR$(ERR)` gives the message for the last error which occurred. Example:

```
TRAP OPEN “\FILE”,A,field1$
IF ERR
  PRINT ERR$(ERR)
  RETURN
ENDIF
```

See also ERR, ERRX$. See the ‘Errors.pdf’ document for full details, including the list of error numbers and messages.

❺ ERRX$

Usage: `x$=ERRX$`

Returns the current extended error message (when an error has been trapped), e.g.

‘Error in *MODULE\PROCEDURE,EXTERN1,EXTERN2,...*’

which would have been presented as an alert if the error had not been trapped. This allows the list of missing externals, missing procedure names, etc. to be found when an error has been trapped by a handler.

See ERR, ERR$. See the ‘Errors.pdf’ document for full details, including the list of error numbers and messages.

ESCAPE OFF

Usage: `ESCAPE OFF`

```
       ...
       ESCAPE ON
```

`ESCAPE OFF` stops Ctrl+Esc on the Series 5 or Psion-Esc on the Series 3c being used to break out of the program when it is running. `ESCAPE ON` enables this feature again.

`ESCAPE OFF` takes effect only in the procedure in which it occurs, and in any sub-procedures that are called. Ctrl+Esc or Psion-Esc is always enabled when a program begins running.

❺ If your program enters a loop which has no logical exit, and `ESCAPE OFF` has been used, you will have to go to the Task list, move to the program name, and select the ‘Close file’ option.

❸ If your program enters a loop which has no logical exit, and `ESCAPE OFF` has been used, you will have to return to the System screen, move to the program name under the RunOpl icon, and press the Delete key.

EVAL

Usage: `d=EVAL(s$)`

Evaluates the mathematical string expression `s$` and returns the floating-point result. `s$` may include any mathematical function or operator. Note that floating-point arithmetic is always performed.

❺ On the Series 5, EVAL runs in the “context” of the current procedure, so globals and externals can be used in `s$`, procedures in loaded modules can be called and the current values of gX and gY, can be used etc. LOCAL variables **cannot** be used in `s$` (because the translator cannot deference them).

For example:

```
DO
  AT 10,5 :PRINT “Calc:”,
  TRAP INPUT n$
  IF n$=“” :CONTINUE :ENDIF
  IF ERR=-114 :BREAK :ENDIF
  CLS :AT 10,4
  PRINT n$;“=”;EVAL(n$)
UNTIL 0
```

See also VAL.

EXIST

Usage: `e%=EXIST(filename$)`

Checks to see that a file exists.

Returns -1 (‘True’) if the file exists and 0 (‘False’) if it doesn’t.

Use this function when creating a file to check that a file of the same name does not already exist, or when opening a file to check that it has already been created:

```
IF NOT EXIST(“CLIENTS”)
  CREATE “CLIENTS”,A,names$
ELSE
  OPEN “CLIENTS”,A,names$
ENDIF
...
```

EXP

Usage: `e=EXP(x)`

Returns e^x - that is, the value of the arithmetic constant e (2.71828...) raised to the power of `x`.

❸ EXT

Usage: `EXT name$`

Gives the file extension of files used by an OPA.

This can only be used between APP and ENDA.

See the ‘Advanced.pdf’ document for more details of OPAs.

❺ OPL application documents do not have file extensions on the Series 5, so this command is not used.

❺ EXTERNAL

Usage: `EXTERNAL` *`variable`*

or `EXTERNAL` *`prototype`*

Required if DECLARE EXTERNAL is specified in the module.

The first usage declares a variable as external. For example, `EXTERNAL screenHeight%`

The second usage declares the *prototype* of a procedure (*`prototype`* includes the final `:` and the argument list). The procedure may then be referred to before it is defined. This allows parameter type-checking to be performed at translate-time rather than at runtime and also provides the necessary information for the translator to coerce numeric argument types. This is reasonable because OPL does not support argument overloading. The same coercion occurs as when calling the built-in keywords.

Following the example of C and C++, you would normally provide a header file declaring prototypes of all the procedures and INCLUDE this header file at the beginning of the module which defines the declared procedures to ensure consistency. The header file would also be INCLUDEd in any other modules which call these procedures. Then you should use DECLARE EXTERNAL at the beginning of modules which include the header file so that the translator can ensure that these procedures are called with correct parameter types or types which can be coerced.

The following is an example of usage of DECLARE EXTERNAL and EXTERNAL:

```
DECLARE EXTERNAL
EXTERNAL myProc%:(i%,l&)
REM or INCLUDE “myproc.oph” that defines all your procedures

PROC test:
  LOCAL i%,j%,s$(10)

  REM j% is coerced to a long integer as specified by the prototype.
  myProc%:(i%,j%)

  REM translator ‘Type mismatch’ error:
  REM string can’t be coerced to numeric type
  myProc%:(i%,s$)

  REM wrong argument count gives translator error
  yProc%:(i%)
ENDP
```

```
PROC myProc%:(i%,l&)
   REM Translator checks consistency with prototype above
   ...
ENDP
```

See DECLARE EXTERNAL.

FIND

Usage: `f%=FIND(a$)`

Searches the current data file (or view on the Series 5) for fields matching `a$`. The search starts from the current record, so use NEXT to progress to subsequent records. FIND makes the next record containing `a$` the current record and returns the number of the record found. Capitals and lower-case letters match.

You can use wildcards:

`?`	matches any single character
`*`	matches any group of characters.

To find a record with a field containing `Dr` and either `BROWN` or `BRAUN`, use:

`F%=FIND("*DR*BR??N*")`

`FIND("BROWN")` will find only those records with a field consisting solely of the string `BROWN`.

You can only search string fields.

See also FINDFIELD.

FINDFIELD

Usage: `f%=FINDFIELD(a$,start%,no%,flags%)`

Like FIND, finds a string, makes the record with this string the current record, and returns the number of this record.

`a$` is the string for which to search: the search will be carried out in `no%` fields in each record, starting at the field with number `start%` (1 is the number of the first field). `start%` and `no%` may refer to string fields only and other types will be ignored. The `flag%` argument specifies the type of search as explained below. If you want to search in all fields, use `start%=1` and for `no%` use the number of fields you used in the OPEN/ CREATE command.

`flags%` should be specified as follows:

search direction	`flags%`
backwards from current record	0
forwards from current record	1
backwards from end of file	2
forwards from start of file	3

❺ 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.

Add 16 to the value of `flag%` given above to make the search *case-dependent*, where case-dependent means that the record will exactly match the search string in case as well as characters. Other wise the search will *case-independent* which means that upper case and lower case characters will match.

See also the ‘Data File Handling’ section of the ‘Database.pdf’ document.

❸ FINDLIB

Usage: `ret%=FINDLIB(var cat%,name$)`

Find DYL category `name$` (including `.DYL` extension) in the ROM. On success returns zero and writes the category handle to `cat%`.

❺ The Series 5 supports use of OPXs only. See the ‘OPX.pdf’ document for further details.

FIRST

Usage: `FIRST`

Positions to the first record in the current data file (or view on the Series 5).

FIX$

Usage: `f$=FIX$(x,y%,z%)`

Returns a string representation of the number `x`, to `y%` decimal places. The string will be up to `z%` characters long.

Example: `FIX$(123.456,2,7)` returns “`123.46`”.

- If `z%` is negative then the string is right-justified for example `FIX$(1,2,-6)` returns “`  1.00`” where there are two spaces to the left of the 1.
- If `z%` is positive then no spaces are added for example `FIX$(1,2,6)` returns “`1.00`”.
- If the number `x` will not fit in the width specified by `z%`, then the string will just be asterisks, for example `FIX$(256.99,2,4)` returns “****”.

See also `GEN$`, `NUM$`, `SCI$`.

❺ FLAGS

Usage: `FLAGS flags%`

Replaces TYPE on the Series 5. `flags%` values as follows:

1 specifies an application which can create files. It will then be included in the list of applications offered when the user creates a new file from the System screen.

2 prevents the application from appearing in the Extras bar. It is very unusual to have this flag set.

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.

FLAGS may only be used within the APP…ENDA construct.

See APP. See also the section on OPAs in the ‘Advanced Topics’ part of the ‘Advanced.pdf’ document.

FLT

Usage: `f=FLT(x&)`

Converts an integer expression (either integer or long integer) into a floating-point number. Example:

```
PROC gamma:(v)
   LOCAL c
   c=3E8
   RETURN 1/SQR(1-(v*v)/(c*c))
ENDP
```

You could call this procedure like this: `gamma:(FLT(a%))` if you wanted to pass it the value of an integer variable without having first to assign the integer value to a floating-point variable.

See also INT and INTF.

FONT

Usage: ❺ `FONT id&,style%`

❸ `FONT id%,style%`

Sets the text window font and style.

❺ Constants for the font UIDs 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.

See ‘The text and graphics windows’ in the ‘Graphics’ section of the ‘GUI.pdf’ document for more details.

FREEALLOC

Usage: ❺ `FREEALLOC pcell&`

❸ `FREEALLOC pcell%`

Frees a previously allocated cell at `pcell&` (`pcell%`).

The number of bytes allocated is restricted to 64K on the Series 3c, while it is not on the Series 5. The input value is therefore a long integer on the Series 5 and an integer on the Series 3c.

❺ See also SETFLAGS if you require the 64K limit to be enforced on the Series 5. If the flag is set to restrict the limit, `pcell&` is guaranteed to fit into a short integer.

GAT

Usage: `gAT x%,y%`

Sets the current position using absolute co-ordinates. `gAT 0,0` moves to the top left of the current drawable.

See also gMOVE.

GBORDER

Usage: `gBORDER flags%,width%,height%`

or `gBORDER flags%`

Draws a one-pixel wide, black border around the edge of the current drawable. If `width%` and `height%` are supplied, a border shape of this size is drawn with the top left corner at the current position. If they are not supplied, the border is drawn around the whole of the current drawable.

`flags%` controls three attributes of the border a shadow to the right and beneath, a one-pixel gap all around, and the type of corners used:

`flags%`	*effect*
1	single pixel shadow
2	removes a single pixel shadow (leaves a gap for single pixel shadow on Series 3c)
3	double pixel shadow
4	removes a double pixel shadow (leaves a gap for double pixel shadow on Series 3c)
$100	one-pixel gap all round
$200	more rounded corners

❺ Constants for the values of 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.

❺ The shadows on the Series 5 will not appear in the same way as shadows on other objects such as dialogs and menu panes appear. To display such shadows on a window, you must specify them when using gCREATE. Hence you should use gCREATE (and gXBORDER) in preference to gBORDER on the Series 5.

You can combine the values to control the three different effects. (1, 2, 3 and 4 are mutually exclusive you cannot use more than one of them.) For example, for rounded corners and a double pixel shadow, use `flags%=$203`.

Set `flags%=0` for no shadow, no gap, and sharper corners.

For example, to de-emphasise a previously emphasised border, use gBORDER with the shadow turned off:

```
gBORDER 3       REM show border
GET
gBORDER 4       REM border off
...
```

See also gXBORDER.

GBOX

Usage: `gBOX width%,height%`

Draws a box from the current position, `width%` to the right and `height%` down. The current position is unaffected.

GBUTTON

Usage: any of

`gBUTTON text$,type%,w%,h%,st%`

❺ `gBUTTON text$,type%,w%,h%,st%,bitmapId&`

❺ `gBUTTON text$,type%,w%,h%,st%,bitmapId&,maskId&`

❺ `gBUTTON text$,type%,w%,h%,st%,bitmapId&,maskId&,layout%`

Draws a 3-D black and grey button at the current position in a rectangle of the supplied width `w%` and height `h%`, which fully encloses the button in all its states. `text$` specifies up to 64 characters to be drawn in the button in the current font and style. You must ensure that the text will fit in the button.

`type%=1` draws a Series 3c button; `type%=2` specifies Series 5.

`state%=0` draws a raised button, `state%=1` a semi-depressed (flat) button and `state%=2` a fully-depressed (sunken) button. On the Series 3c, an error is raised if the current window has no grey plane.

❺ On the Series 5, there is added support so that bitmaps may be used on buttons. Three extra optional arguments can be passed which give the bitmap ID, the mask ID and the layout for the button respectively. `maskId%` can be 0 to specify no mask.

The following constants should be used for `layout%` to specify relative positions of the text and icon on a button,

position of text	`layout%`
right	0
bottom	1
top	2
left	3

The following constants can be added to the values above to specify how a button's excess space is to be allocated,

share	0
to text	$10
to picture	$20

Constants for all these layout types and for the button states 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.

When the layout is such that the text is at the top or the bottom, then text and picture are centred vertically and horizontally in the space allotted to them. If the layout has text to the left or right, then the text is left aligned in the space allotted to it and the picture is right or left aligned respectively. Both text and picture are centred vertically in this case.

Examples:

`layout%`	*description*
`$13`	creates a button with text on the left and left aligned in any excess space.
`$20`	creates a button with text on the right and the picture left aligned in any excess space.
`$10`	creates a standard toolbar button, putting the text on the right.

For a picture only with no text use `text$=“”`.

‘Invalid arguments’ error is raised if you use OPL windows for gBUTTON. Read-only bitmaps may also be loaded using the Bitmap OPX. See the ‘OPX.pdf’ document for more details of how to do this.

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.

❺ GCIRCLE

Usage: `gCIRCLE radius%`

or `gCIRCLE radius%,fill%`

Draws a circle with the centre at the current position in the current drawable. If the value of `radius%` is negative then no circle is drawn.

If `fill%` is supplied and if `fill%`<>0 then the circle is filled with the current pen colour.

See gELLIPSE, gCOLOR.

GCLOCK

Usage: any of

```
gCLOCK ON/OFF
gCLOCK ON,mode%
gCLOCK ON,mode%,offset&
gCLOCK ON,mode%,offset&,format$
gCLOCK ON,mode%,offset&,format$,font%
gCLOCK ON,mode%,offset&,format$,font%,style%
```

❸ **Note: `offset&` is replaced by `offset%` and `font&` by `font%` on the Series 3c.**

Displays or removes a clock showing the system time. The current position in the current window is used. Only one clock may be displayed in each window.

`mode%` controls the type of clock.

Values are:

	6	black and grey medium, system setting
	7	black and grey medium, analog
	8	second type medium, digital
	9	black and grey extra large
❸	10	formatted digital (described below)
❺	11	formatted digital (described below)

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 Appendix E in the ‘Appends.pdf’ document for a listing of it.

❸ On the Series 3c, modes 1 to 5 are provided for Series 3 compatibility, and produce small (digital), medium (system setting), medium (analog), medium (digital), and large (analog) clocks respectively.

You can also OR the mode with any of these to create the following effects:

`$10`	shows the date in all except the extra large and formatted clocks
`$20`	shows seconds in small digital, large analog, black and grey medium analog and extra large clocks
`$40`	shows am/pm in small digital and black medium clocks only.
`$80`	specifies that a clock is to be drawn in the grey plane (only for clocks that do not contain both black and grey: i.e. all except the black and grey, medium, analog clock and the extra large clock).

✍ It is possible to draw clocks that include grey in windows that have no grey plane.

❺ On the Series 5, there are additional features on the basic clocks which partially replace these effects. The digital clock (`mode%=8`) automatically displays the day of the week and day of the month below the time. The extra large analog clock (`mode%=9`) automatically displays a second hand.

⚠ Do not use gSCROLL to scroll the region containing a clock. When the time is updated, the old position would be used. The whole window may, however, be moved using gSETWIN.

Digital clocks display in 24-hour or 12-hour mode according to the system-wide setting.

`offset&` specifies an offset in minutes from the system time to the time displayed. This allows you to display a clock showing a time other than the system time.

❺ On the Series 5, a flag, which has the value $100, may be ORed with `mode%` so that `offset&` may be specified in seconds rather than minutes. The offset is a long integer to enable a whole day to be specified when the offset is in seconds.

If these arguments are not supplied, `mode%` is taken as 1, and `offset&` as 0.

❺ The system setting for the clock type (i.e. digital or analog) can be changed by an OPL program using the procedure LCSETCLOCKFORMAT: in Date OPX This is function should be used to implement, for example, tapping a toolbar clock to change its type as in the built-in Series 5 applications. See the ‘OPX.pdf’ document for full details of this procedure.

`format$`, `font%` and `style%` are used only for formatted digital clocks (`mode%` 10 on the Series 3c and 11 on the Series 5). The values for `font&` and `style%` are as for gFONT and gSTYLE. The default font for gCLOCK is the system font. The default style is normal (0).

For the formatted digital clock, a format string (up to 255 characters long) specifies how the clock is to be displayed. The format string contains a number of format specifiers in the form of a `%` followed by a letter. (Upper or lower case may be used.)

❺ For the Series 5, the format string may contain the following symbols to obtain the required effects:

`%%` Insert a single % character in the string

`%*` Abbreviate following item. (The asterisk should be inserted between the % and the number or letter, e.g. %*1). In most cases this amounts to omitting any leading zeros, for example if it is the first of the month "%F %*M" will display as 1 rather than 01.

`%:`*n* Insert a system time separator character. *n* is an integer between zero and three inclusive which indicates which time separator character is to be used. For European time settings, only *n*=1 and *n*=2 are used, giving the hours/minutes separator and minutes/seconds separator respectively.

`%/`*n* Insert a system date separator character. *n* is an integer between zero and three inclusive which indicates which date separator character is to be used. For European time settings, only *n*=1 and *n*=2 are used, giving the day/month separator and month/year separator respectively.

`%1` Insert the first component of a three component date (i.e. a date including day, month and year) where the order of the components is determined by the system settings. The possibilities are: dd/mm/yyyy, (European), mm/dd/yyyy (American), yyyy/mm/dd (Japanese).

`%2` Insert the second component of a three component date where the order has been determined by the system settings. See %1.

`%3` Insert the third component of a three component date where the order has been determined by the system settings. See %1.

`%4` Insert the first component of a two component date (i.e. a date including day and month only) where the order has been determined by system settings. The possibilities are: dd/mm, (European), mm/dd (American), mm/dd (Japanese).

`%5` Insert the second component of a two component date where the order has been determined by the system settings. See %4.

`%A` Insert am or pm according to the current language and time of day. Text is printed even if 24 hour clock is in use. Text may be specified to be printed before or after the time, and a trailing or leading space as appropriate will be added. The abbreviated version (%*A) removes this space. Optionally, a minus or plus sign may be inserted between the % and the A. This operates as follows: %-A causes am/pm text to be inserted only if the system setting of the am/pm symbol position is set to display **before** the time. Similarly, %+A causes am/pm text to be inserted only if the system setting of the am/pm symbol is set to display **after** the time. No am/pm text will be inserted before the time if a + is inserted in the string. For example you could use, "%-A%H%:1%T%+A" to insert the am/pm symbol **either** before or after the time, according to the system setting. %+A and %-A cannot be abbreviated.

`%B` As %A, except that the am/pm text is only inserted if the system clock setting is 12 hour. (This should be used in conjunction with %J.)

`%D` Insert the two-digit day number in month (in conjunction with %1 etc.).

`%E` Insert the day name. Abbreviation is language specific (3 letters in English).

`%F` Use this at the beginning of a format string to make the date/time formatting independent of the system setting. This fixes the order of the following day/month/year component(s) in their given order, removing the need to use %1 to %5, allowing individual components of the date to be printed. (No abbreviation.)

`%H` Insert the two-digit hour component of the time in 24 hour clock format.

`%I` Insert the two-digit hour component of the time in 12 hour clock format. Any leading zero is automatically suppressed, regardless of whether an asterisk is inserted or not.

`%J` Insert the two-digit hour component of time in either 12 or 24 hour clock format depending on the corresponding system setting. When the clock has been set to 12 hour format, the hour’s leading zero is automatically suppressed regardless of whether an asterisk has been inserted between the % and J.

`%M` Insert the two-digit month number (in conjunction with %1 etc.).

`%N` Insert the month name (in conjunction with %1 etc.). When using system settings (i.e. not using %F) this causes all months following %N in the string to be written in words. When using fixed format (i.e. when using %F) %N may be used alone to insert a month name. Abbreviation is language specific (3 letters in English).

`%S` Insert the two-digit second component of the time.

`%T` Insert the two-digit minute component of the time.

`%W` Insert the two-digit week number in year, counting the first (part) week as week 1.

`%X` Insert the date suffix. When using system settings (i.e. not using %F), this causes a suffix to be put on any date following %X in the string. When using fixed format (i.e. using %F), %X following any date appends a suffix for that particular date. Cannot be abbreviated.

`%Y` Insert the four digit year number (in conjunction with %1 etc.). The abbreviation is the last two digits of the year.

`%Z` Insert the three digit day number in year.

Some examples of the use of these format strings are as follows. The example use is 1:30:05 pm on Wednesday, 1st January 1997, with the system setting of European dates and with am/pm after the time:

1. `“%-A%I:%T:%S%+A”` will print the time in 12 hour clock, including seconds, with the am/pm either inserted before or after the time, depending on the system setting. So the example time would appear as, `1:30:05 pm`.
2. `“%F%E %*D%X %N %Y”` will print the day of the week followed by the date with suffix, the month as a word and the year. For example, `Wednesday 1st January 1997`.
3. `“%E %D%X%N%Y %1 %2 %3”` will use the locale setting for ordering the elements of the date, but will use a suffix on the day and the month in words. For example, `Wednesday 01st January 1997`.
4. `“%*E %*D%X%*N%*Y %1 %2 ‘%3”` will be similar to 3., but will abbreviate the day of the week, the day, the month and the year, so the example becomes `“Wed 1st Jan 97”`.

5. `“%M%Y%D%1%/0%2%/0%3”` will appear as `01/01/1997`. This demonstrates that the ordering of the `%D`, `%M` and `%Y` is irrelevant when using locale-dependent formatting. Instead the ordering of the date components is determined by the order of the `%1`, `%2`, and `%3` formatting commands.

`style%` may take any of the values used to specify gSTYLE, other than 2 (underlined).

A note should also be made that a ‘General Failure’ error will result if you attempt to use an invalid format. Invalid formats include using `%:` and `%/` followed by 0 or 3 when in European locale setting (when these separators are without meaning) and using `%+` and `%-` followed by characters other than A or B.

❸ The format string for the Series 3c may be defined in a similar manner to the Series 5, although generally less functionality is available. Any item may be abbreviated by using a `*` after the `%`. For example, `%*T` at 11:05 pm abbreviates `05` to `5`. In the following list of specifiers, those which produce numbers will do so without any leading zero if you use `%*` instead of `%`. Other abbreviations are marked:

`%%` Insert a single % character in the string.

`%:, %/` Insert a system time, date separator character.

`%A` Insert am or pm text, according to the system time. (Abbreviation: 1st letter only)

`%D, %W, %M` Insert the day, week, month number as two digits, in the range 01-31, 01-53 and 01-12, respectively

`%E, %N` Insert the day, month name. (Abbreviation: language dependent - first 3 letters in English)

`%H, %I` Insert the hour in 24-hour, 12-hour format, in the range 00-23 and 01-12, respectively

`%S, %T` Insert the seconds, minutes, in the range 00-59.

`%X` Insert the suffix string for day number, e.g. st in ‘1st’, nd in ‘2nd’

`%Y` Insert the year as a four digit number (Abbreviation: discards the century, i.e. last two digits)

`%1, %2, %3` Insert the day, month, year ordered according to the system setting. E.g. European setting is day/month/year, so %1=%D, %2=%M, %3=%Y. So to display a date in correct format use “%1/%2/%3”. (Abbreviation: see %G, %P, %U)

`%4, %5` Insert the day, month as ordered in the system setting.

`%F, %O` Toggles days, months (displayed by %1, %2 and %3) between numeric and name formats. On 9th March 1993, with European date type, “%1%F%1%F%1” gives 09Tuesday09.

`%G, %P, %U` Toggles %1, %2 and %3 between long form and abbreviation. On 9th March 1993, with European date type, “%F%1%G%1%G%1” gives TuesdayTueTuesday.

`%L` Toggles the suffix on the day number for %1, %2, %3 (in numeric form only). On 9th March 1993, with European date type, “%G%1%L%1%L%1” gives 99th9.

`%6, %7` Inserts the hour and am/pm text according to the system setting. With am-pm format, %6=%I and %7=%A. With 24-hour format, %6=%H and %7 gives no ‘am/pm’ characters.

For example, the format strings

`“%H, m:%T”` at 11:05 pm, displays a running clock as `h:23, m:05`.

`“%1%/%2%/%3”` automatically generates a clock with day, month and year in the order as selected in the Time application.

`“%4%/%5”` gives a clock with just day and month in selected order.

`“%6%:%T%:%S%7”` gives a clock with hour, minute and second automatically conforming to the system configuration.

Note that for those specifiers that toggle between two different options (e.g. `%F`), the state of toggle is remembered only within one format string and not from one string to the next - i.e. the toggle state is restored to the default setting when displaying a new clock.

As a final example, assuming that the settings in the Time application are for ‘day/month/year’ date format, ‘am-pm’ time format and ‘:’ time separator and that the time is 11:30:05 pm on 9th March 1993, `“%G%L%P%O%*E, %1 %2 %3 %6%:%T:%S%”` generates `Tue, 9th Mar 1993 11:30:05pm`. With the same setup except for ‘month/day/year’ date format in ‘24-hour’ mode, the same string generates `Tue, Mar 9th 1993 23:30:05`.

GCLOSE

Usage: `gCLOSE id%`

Closes the specified drawable that was previously opened by gCREATE, gCREATEBIT or gLOADBIT.

If the drawable closed was the current drawable, the default window (ID=1) becomes current.

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

GCLS

Usage: `gCLS`

Clears the whole of the current drawable and sets the current position to 0,0, its top left corner.

❺ GCOLOR

Usage: `gCOLOR red%,green%,blue%`

Sets the pen colour of the current window. The `red%,green%,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, ranging from black (0) to white (255).

GCOPY

Usage: `gCOPY id%,x%,y%,w%,h%,mode%`

Copies a rectangle of the specified size (width `w%`, height `h%`) from the point `x%,y%` in drawable `id%`, to the current position in the current drawable.

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.

As this command can copy both set and clear pixels, the same modes are available as when displaying text. Set `mode%` = 0 for set, 1 for clear, 2 for invert or 3 for replace. 0, 1 and 2 act only on set pixels in the pattern; 3 copies the entire rectangle, with set and clear pixels.

The current position is not affected in either window.

❸ gCOPY is affected by the setting of gGREY (in the **current window**) as follows: with `gGREY 0` it copies black to black; with `gGREY 1` it copies grey to grey, or black to grey if source is black only; with `gGREY 2` it copies grey to grey and black to black, or black to both if source is black only.

GCREATE

Usage: any of

```
id%=gCREATE(x%,y%,w%,h%,v%)
```

❸
```
id%=gCREATE(x%,y%,w%,h%,v%,grey%)
```

❺
```
id%=gCREATE(x%,y%,w%,h%,v%,flags%)
```

Creates a window with specified position and size (width `w%`, height `h%`), and makes it both current and foreground. Sets the current position to 0,0, its top left corner. If `v%` is 1, the window will immediately be visible; if 0, it will be invisible.

Returns `id%` which identifies this window for other keywords.

❺ `flags%` specifies the graphics mode to use and shadowing on the window. By default the graphics mode is 2-colour and there is no shadow.

The least significant 4 bits of `flags%` gives the colour-mode as before 0 (2 colour-mode), 1 (4 colour-mode), 2 (16 colour-mode).

The next 4 bits may be set to specify the shadowing on the window. If 0, the window has no shadow. The next 4 bits give the shadow height relative to the window behind it (a height of *N* units gives a shadow of *N*×2 pixels).

The `flags%` argument is most easily specified in hexadecimal:

`flags%`	*description*
`$412`	16 colour-mode ($2), shadowed window ($1), with height 4 units ($4) above the previous window with a shadow of 8 pixels.
`$010`	2 colour-mode (black and white) shadowed window at the same height as the previous window.
`$101`	4 colour mode window with no shadow (height ignored if shadow disabled).
`$111`	4 colour mode window with shadow of 1 unit above window behind, i.e. 2 pixel shadow.

Constants for specifying various of the arguments taken by gCREATE are given 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.

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 If `grey%` is not given or is 0, the window will not have a grey plane. If `grey%` is 1, it will have one.

See also gCLOSE, gGREY, DEFAULTWIN.

GCREATEBIT

Usage: `id%=gCREATEBIT(w%,h%)`

or ❺ `id%=gCREATEBIT(w%,h%,mode%)`

Creates a bitmap with the specified width and height, and makes it the current drawable. Sets the current position to 0,0, its top left corner.

Returns `id%` which identifies this bitmap for other keywords.

❺ gCREATEBIT may be used with an optional third parameter which specifies the graphics mode of the bitmap to be created. The values of these are as given in gCREATE. By default the graphics mode of a bitmap is 2-colour.

Note that 64 drawables may be open at any time. Although, as mentioned above, using a large number of windows should be avoided in practice, you can sensibly use as many bitmaps as you need up to the maximum.

See also gCLOSE,gCREATE.

❸ GDRAWOBJECT

Usage: `gDRAWOBJECT type%,flags%,w%,h%`

Draws the scaleable graphics object specified by `type%`, scaled to fit in the rectangle with top left at the current graphics cursor position and with the specified width `w%` and height `h%`.

The Series 3c has only one object type (set `type%=0`) a ‘lozenge’. This is a 3-D rounded box lit from the top left, with a shadow at bottom right and a grey body. (For an example, see the text ‘City’ in the top left of the World application.)

For `type%=0`, `flags%` specifies the corner roundness:

0 for normal roundness

1 for more rounded

2 for a single pixel removed from each corner.

An error is raised if the current window has no grey plane.

❺ GELLIPSE

Usage: `gELLIPSE hRadius%,vRadius%`

or `gELLIPSE hRadius%,vRadius%,fill%`

Draws an ellipse with the centre at the current position in the current drawable. `hRadius%` is the horizontal distance in pixels from the centre of the ellipse to the left (and right) of the ellipse. `vRadius%` is the vertical distance from the centre of the ellipse to the top (and bottom). If the length of either radius is less than zero, then no ellipse is drawn.

If `fill%` is supplied and if `fill%<>0` then the ellipse is filled with the current pen colour.

See gCIRCLE, gCOLOR.

GEN$

Usage: `g$=gen$(x,y%)`

Returns a string representation of the number `x`. The string will be up to `y%` characters long.

Example `GEN$(123.456,7)` returns "`123.456`" and `GEN$(243,5)` returns "`243`".

- If `y%` is negative then the string is right-justified - for example `GEN$(1,-6)` returns "   `1`" where there are five spaces to the left of the 1.
- If `y%` is positive then no spaces are added for example `GEN$(1,6)` returns "`1`".
- If the number x will not fit in the width specified by y%, then the string will just be asterisks, for example GEN$(256.99,4) returns "`****`".

See also FIX$, NUM$, SCI$.

GET

Usage: `g%=GET`

Waits for a key to be pressed and returns the character code for that key.

For example, if the A key is pressed with Caps Lock off, the integer returned is 97 (a), or 65 (A) if A was pressed with the Shift key down.

The character codes of special keys, such as Pg Dn, are given in Appendix D in the 'Appends.pdf' document..

You can use KMOD to check whether modifier keys (Shift, Control, Psion (on the Series 3c), Fn (on the Series 5) and Caps Lock) were used.

See Appendix D in the 'Appends.pdf' document for the full character set for the Series 5. For the Series 3c, see the User Guide.

See also KEY.

GET$

Usage: `g$=GET$`

Waits until a key is pressed and then returns which key was pressed, as a string.

For example, if the A key is pressed in lower case mode, the string returned is `"a"`.

You can use KMOD to check whether any modifier keys (Shift, Control, Psion (on the Series 3c), Fn (on the Series 5) and Caps Lock) were used.

See also KEY$.

GETCMD$

Usage: `w$=GETCMD$`

Returns new command-line arguments to an OPA, after a “change files” or “quit” event has occurred. The first character of the returned string is `“C”`, `“O”` or `“X”`. If the return is `“C”` or `“O”`, the rest of the string is a filename.

The first character has the following meaning:

`“C”` - close down the current file, and create the specified new file,

`“O”` - close down the current file, and open the specified existing file,

`“X”` - close down the current file (if any) and quit the OPA.

❺ Constants for these return 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.

You can only call GETCMD$ once for each system message.

See the ‘Advanced.pdf’ document for more details of OPAs.

See also CMD$.

❺ GETDOC$

Usage: `docname$=GETDOC$`

Returns the name of the current document.

See also SETDOC.

GETEVENT

Usage: `GETEVENT var a%()`

Waits for an event to occur. Returns with `a%()` specifying the event. The data returned in `a%()` depends on the type of event that occurred. If the event is a key-press, `(a%(1) AND $400)` is guaranteed to be zero. For other events `(a%(1) AND $400)` is guaranteed to be non-zero.

If a key has been pressed:	`a%(1)`	keycode (as for GET)
	`a%(2) AND $00ff`	modifier (as for KMOD)
	`a%(2)/256`	auto-repeat count (ignored by GET)
If a program has moved to		
foreground:	`a%(1)= $401`	
background:	`a%(1)= $402`	
If the machine has switched on:	`a%(1)= $403`	
If the Psion wants an OPA to change files or exit:	`a%(1)= $404`	
If the date changes:	`a%(1)= $405`	

❺ Note that date change events are not detected by the Series 5.

Also note that GETEVENT responds to all events to which GETEVENT32 responds, including pen events. The same codes are written to `ev%(1)` as `GETEVENT32 ev&()` writes to `ev&(1)`, but the array elements `ev%(2)` to `ev%(6)` are only set as detailed above.

⚠ It is strongly recommended that you use GETEVENT32 rather than GETEVENT if you are using the Series 5. GETEVENT is supported only for backward compatibility and cannot be used to handle pen events in a satisfactory way.

For a key-press event, the modifier is returned in `a%(2)` and is not returned by KMOD.

✎ If a non-key event such as ‘foreground’ occurs while a keyboard keyword such as GET, INPUT, MENU or DIALOG is being used, the event is discarded. So GETEVENT must be used if non-key events are to be monitored. If you need to use these keywords in OPAs, use `LOCK ON` / `LOCK OFF` around them, so that the System screen won’t send messages to switch files or shutdown while the application cannot respond.

The array (or string of integers) **must** be at least 6 integers long.

See also TESTEVENT, GETCMD$, GETEVENT32.

❺ GETEVENT32

Usage: `GETEVENT32 ev&()`

Gets all event types handled by `GETEVENT ev%()` and additionally pointer (pen) events. The latter are too large to fit into the array of integers provided for `GETEVENT ev%()`. `ev&()` must have at least 16 elements.

All events return a 32-bit time stamp. The window ID mentioned below refers to the value returned by the gCREATE keyword. The modifier values and scancode values for a keypress (which specify a location on the keyboard) are given in the ‘Advanced.pdf’ document and Appendix D in the ‘Appends.pdf’ document.

GETEVENT32 returns more information than GETEVENT, as listed below:

If a key has been pressed:	`(ev&(1) AND &400)=0`	
	`ev&(1)`	keycode
	`ev&(2)`	time stamp
	`ev&(3)`	scan code
	`ev&(4)`	modifier
	`ev&(5)`	repeat

Note that unlike the repeat for GETEVENT, the repeat for GETEVENT32 is strictly a repeat, i.e. if there is only one keypress, then the value of `ev&(5)` is 0.

For all the other event types, `ev&(1)` is greater than &400:

If the program has moved to foreground:	`ev&(1)=&401`	
	`ev&(2)`	time stamp
If the program has moved to background:	`ev&(1)=&402`	
	`ev&(2)`	time stamp
If the machine is switched on:	`ev&(1)=&403`	
	`ev&(2)`	time stamp

Note that this event is **not** enabled unless the appropriate flag is set (by default it is not): see SETFLAGS.

If the Series 5 wants the OPL application to switch files or exit:	`ev&(1)=&404`	

If this event is received, GETCMD$ should be called to find out what action should be taken: see GETCMD$.

If a key is pressed down:	`ev&(1)=&406`	
	`ev&(2)`	time stamp
	`ev&(3)`	scan code
	`ev&(4)`	modifiers
If a key is released:	`ev&(1)=&407`	
	`ev&(2)`	time stamp
	`ev&(3)`	scan code
	`ev&(4)`	modifiers
If a pen event occurs:	`ev&(1)=&408`	
	`ev&(2)`	time stamp
	`ev&(3)`	window ID
	`ev&(4)`	pointer type (see below)
	`ev&(5)`	modifiers
	`ev&(6)`	x-co-ordinate
	`ev&(7)`	y-co-ordinate
	`ev&(8)`	x-co-ordinate relative to parent window
	`ev&(9)`	y-co-ordinate relative to parent window

For pen events, `ev&(4)` has one of the following values:

0	pen down	1	pen up	6	drag

If a pen enters contact with the screen:	`ev&(1)=&409`	
	`ev&(2)`	time stamp
	`ev&(3)`	window ID
If a pen exits contact with the screen:	`ev&(1)=&40A`	
	`ev&(2)`	time stamp
	`ev&(3)`	window ID

Constants for the array subscripts and the return 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.

Some pointer events, and pointer enters and exits, can be filtered out to avoid being swamped by unwanted event types. See POINTERFILTER.

Note that for other unknown events, `ev&(1)` contains &1400 added to the code returned by the window server. `ev&(2)` is the timestamp and `ev&(3)`is the window ID, and the rest of the data returned by the window server is put into `ev&(4)`, `ev&(5)`, etc.

If a non-key event such as ‘foreground’ occurs while a keyboard keyword such as GET, INPUT, MENU or DIALOG is being used, the event is discarded. So GETEVENT must be used if non-key events are to be monitored. If you need to use these keywords in OPAs, use `LOCK ON` / `LOCK OFF` around them, so that the System screen won’t send messages to switch files or shutdown while the application cannot respond.

See also GETEVENT, GETEVENTA32.

❺ GETEVENTA32

Usage: `GETEVENTA32 status%,ev&()`

Asynchronous version of GETEVENT32. GETEVENTA32 returns the same codes to the array `ev&()` as GETEVENT32.

See GETEVENTC, GETEVENT32, GETEVENT. See also ‘I/O functions and commands’ in the ‘Advanced.pdf’ document for details of asynchronous I/O functions.

❺ GETEVENTC

Usage: `GETEVENTC(var stat%)`

Cancels the previously called GETEVENTA32 function with status `stat%`. Note that GETEVENTC consumes the signal (unlike IOCANCEL), so IOWAITSTAT should not be used after GETEVENTC.

See the ‘Advanced.pdf’ document for details.

❸ GETLIBH

Usage: `cat%=GETLIBH(num%)`

Convert a category number `num%` to a handle. If `num%` is zero, gets the handle for OPL.DYL.

GFILL

Usage: `gFILL width%,height%,gMode%`

Fills a rectangle of the specified size from the current position**, according to the graphics mode specified**.

The current position is unaffected.

GFONT

Usage: ❺ `gFONT fontUid&`

❸ `gFONT fontId%`

Sets the font for current drawable to `fontId%`. The font may be one of the predefined fonts in the ROM or a user-defined font. See the ‘Graphics’ section of the ‘GUI.pdf’ document for more details of fonts.

❺ Constants for the font UIDs 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.

User-defined fonts must first be loaded by gLOADFONT, then the font UIDs of the loaded fonts may be used with gFONT. Note that this is **not** the ID returned by gLOADFONT (which is the font file ID), but the UID defined in the font file itself.

❸ User-defined fonts must first be loaded by gLOADFONT, which returns the `fontId%` which may be used with gFONT.

See also gLOADFONT, FONT.

GGMODE

Usage: `gGMODE mode%`

Sets the effect of all subsequent drawing commands gLINEBY, gBOX etc. on the current drawable.

`mode%`	*pixels will be:*
0	set
1	cleared
2	inverted

❺ Constants for the mode 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.

When you first use drawing commands on a drawable, they set pixels in the drawable. Use gGMODE to change this. For example, if you have drawn a black background, you can draw a white box outline inside it with either `gGMODE 1` or `gGMODE 2`, followed by `gBOX`.

GGREY

Usage: `gGREY mode%`

❺ Changes the pen colour between black and grey. `mode%` has the following effects:

`mode%=1` sets the foreground colour of the current drawable to light grey. This is the same colour as would be achieved by using `gCOLOR $aa,$aa,$aa`.

`mode%` of any other value sets the foreground colour to black (the default).

❸ Controls whether all subsequent graphics drawing and graphics text **in the current window** draw to the grey plane, the black plane or to both.

`mode%=0` for black plane only (default)

`mode%=1` for grey plane only

`mode%=2` for both planes

It is helpful to think of the black plane being in front of the grey plane, so a pixel set in both planes will appear black. See the ‘Graphics’ section of the ‘GUI.pdf’ document for details.

To enable the use of grey in the default window (ID=1) use `DEFAULTWIN 1` at the start of your program. If grey is required in other windows you must **create** the windows with a grey plane using gCREATE.

gGREY cannot be used with bitmaps which have only one plane.

See also DEFAULTWIN and gCREATE.

GHEIGHT

Usage: `height%=gHEIGHT`

Returns the height of the current drawable.

GIDENTITY

Usage: `id%=gIDENTITY`

Returns the ID of the current drawable.

The default window has ID=1.

❸ GINFO

Usage: `gINFO var i%()`

Gets general information about the current drawable and about the graphics cursor (whichever window it is in). The information is returned in the array `i%()` which must be at least 32 integers long.

The information is about the drawable in its current state, so e.g. the font information is for the current font in the current style.

The following information is returned:

`i%(1)`	lowest character code
`i%(2)`	highest character code
`i%(3)`	height of font
`i%(4)`	descent of font
`i%(5)`	ascent of font
`i%(6)`	width of ‘0’ character
`i%(7)`	maximum character width
`i%(8)`	flags for font (see below)
`i%(9-17)`	name of font
`i%(18)`	current graphics mode (gGMODE)
`i%(19)`	current text mode (gTMODE)
`i%(20)`	current style (gSTYLE)
`i%(21)`	cursor state (ON=1,OFF=0)
`i%(22)`	ID of window containing cursor (-1 for text cursor)
`i%(23)`	cursor width
`i%(24)`	cursor height
`i%(25)`	cursor ascent
`i%(26)`	cursor x position in window
`i%(27)`	cursor y position in window
`i%(28)`	1 if drawable is a bitmap
`i%(29)`	cursor effects
`i%(30)`	gGREY setting

`i%(31)`	reserved (window server ID of drawable)
`i%(32)`	reserved

`i%(8)` specifies a combination of the following font characteristics:

value	*meaning*
1	font uses standard ASCII characters (32-126)
2	font uses Code Page 850 characters (128-255)
4	font is bold
8	font is italic
16	font is serifed
32	font is mono-spaced
$8000	font is stored expanded for quick drawing

Use `PEEK$(ADDR(i%(9)))` to read the name of the font as a string.

If the cursor is on (`i%(21)`=1), it is visible in the window identified by `i%(22)`.

`i%(29)` has bit 0 set (`i%(29) AND 1`) if the cursor is obloid, bit 1 set (`i%(29) AND 2`) if not flashing, and bit 2 set (`i%(29) AND 4`) if grey.

If the cursor is off (`i%(21)`=0), or is a text cursor (`i%(22)=-1`), `i%(23)` to `i%(27)` and `i%(29)`should be ignored.

❺ GINFO32

Usage: `gINFO32 var i&()`

Gets general information about the current drawable and about the graphics cursor (whichever window it is in). This replaces `gINFO` because the information available has changed. `i&()` must have 48 elements (although elements 37 to 48 are currently unused). The same information is returned to the array elements as for gINFO except for the following,

`i&(1)`	reserved
`i&(2)`	reserved
`i&(9)`	the font UID as used in gFONT
`i&(10-17)`	unused
`i&(30)`	graphics colour-mode of current window
`i&(31)`	gCOLOR `red%` of foreground
`i&(32)`	gCOLOR `green%` of foreground
`i&(33)`	gCOLOR `blue%` of foreground
`i&(34)`	gCOLOR `red%` of background
`i&(35)`	gCOLOR `green%` of background
`i&(36)`	gCOLOR `blue%` of background

Additionally note that on the Series 5, `i&(8)=2` means that Code Page 1252 is used (rather than Code Page 850) and also that there is no obloid cursor, so bit 0 will never be set in `i&(29)`.

See also gINFO, gFONT, gCOLOR, gCREATE.

GINVERT

Usage: `gINVERT width%,height%`

Inverts the rectangle `width%` to the right and `height%` down from the cursor position, except for the four corner pixels.

GIPRINT

Usage: `gIPRINT str$,c%`

or `gIPRINT str$`

Displays an information message for about two 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.

If `c%` is given, it controls the corner in which the message appears:

`c%`	*corner*
0	top left
1	bottom left
2	top right
3	bottom right (default)

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.

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 “”`.

GLINEBY

Usage: `gLINEBY dx%,dy%`

Draws a line from the current position to a point `dx%` to the right and `dy%` down. Negative `dx%` and `dy%` mean left and up respectively.

❺ The Series 5 never draws the end point, so 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 For horizontal lines, the line includes the pixel with the lower x co-ordinate and excludes the pixel with the higher x co-ordinate. Similarly for vertical lines, the line includes the pixel with the lower y co-ordinate and excludes the pixel with the higher y co-ordinate. For oblique lines (where the x and y co-ordinates change), the line is drawn minus one or both end points.

For all machines, the current position moves to the end of the line drawn.

`gLINEBY 0,0` sets the pixel at the current position.

See also gLINETO, gPOLY.

GLINETO

Usage: `gLINETO x%,y%`

Draws a line from the current position to the point `x%,y%`. The current position moves to `x%,y%`.

❺ The Series 5 never draws the end point, so for `gLINETO x%,y%`, point `x%,y%` is not drawn

Note, however, that OPL specially plots the point when the start and end-point coincide.

❸ For horizontal lines, the line includes the pixel with the lower x co-ordinate and excludes the pixel with the higher x co-ordinate. Similarly for vertical lines, the line includes the pixel with the lower y co-ordinate and excludes the pixel with the higher y co-ordinate. For oblique lines (where the x and y co-ordinates change), the line is drawn minus one or both end points.

To plot a single point on all machines, use gLINETO to the current position (or `gLINEBY 0,0`).

See also gLINEBY, gPOLY.

GLOADBIT

Usage: any of

```
id%=gLOADBIT(name$,write%,index%)
id%=gLOADBIT(name$,write%)
id%=gLOADBIT(name$)
```

Loads a bitmap from the named bitmap file and makes it the current drawable. Sets the current position to 0,0, its top left corner.

❺ gLOADBIT does not add a default filename extension to the input argument name.

Note that on the Series 5, gLOADBIT loads EPOC Picture files, which are naturally in the same file format that is saved by gSAVEBIT. EPOC Picture files can also be generated by exporting files created by the Sketch application. These are called multi-bitmap files (MBMs), though often containing just one bitmap as in the case of gSAVEBIT or Sketch files, and are often given an extension .MBM.

❸ If `name$` has no file extension, `.PIC` is used.

The bitmap is kept as a local copy in memory.

Returns `id%` which identifies this bitmap for other keywords.

`write%`=0 sets read-only access. Attempts to write to the bitmap in memory will be ignored, but the bitmap can be used by other programs without using more memory. `write%`=1 allows you to write to and re-save the bitmap. This is the default case.

❺ Constants for the values of `write%` 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 bitmap files which contain more than one bitmap, `index%` specifies which one to load. For the first bitmap, use `index%`=0, which is also the default value.

❸ Bitmap files saved with gSAVEBIT have only one bitmap if they are saved from an in-memory bitmap rather than from a window. Saving a black/grey/white window on the Series 3c saves two planes black to `index%=0` and grey to `index%=1`.

See also gCLOSE.

GLOADFONT

Usage: ❺ `fileId%=gLOADFONT(file$)`

❸ `fontId%=gLOADFONT(name$)`

Loads the user-defined font. This is done differently between machines as follows:

❺ Loads the user-defined fonts specified in the file `file$` and returns the file ID of the font file, 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 file you need to use their published UIDs which will be defined in the font file itself, for example:

```
fileId%=gLOADFONT(“Music1”)
gFONT KMusic1Font1&
...
gUNLOADFONT fileId%
```

❸ Loads the user-defined font `name$` and returns a font ID which can be used with gFONT to make the current drawable use this font. If `name$` does not contain a file extension, `.FON` is used.

gFONT itself is very efficient, so you should normally load all required fonts at the start of a program.

Note that the built-in fonts are automatically available, and do not need loading.

See gUNLOADFONT.

GLOBAL

Usage: `GLOBAL` *`variables`*

Declares variables to be used in the current procedure (as does the LOCAL command) *and* (unlike LOCAL) in any procedures called by the current procedure, or procedures called by them.

The variables may be of 4 types, depending on the symbol they end with:

- Variable names not ending with `$`, `%`, `&` or `()` are floating-point variables, for example price, `x`
- Those ending with a `%` are integer variables, for example `x%`, `sales92%`

- Those ending with an & are long integer variables, for example `x&`, `sales92&.`

- Those ending with a `$` are string variables. String variable names must be followed by the maximum length of the string in brackets for example `names$(12)`, `a$(3)`

Array variables have a number immediately following them in brackets which specifies the number of elements in the array. Array variables may be of any type, for example: `x(6),y%(5),f$(5,12),z&(3)`.

When declaring string arrays, you must give two numbers in the brackets. The first declares the number of elements, the second declares their maximum length. For example `surname$(5,8)` declares five elements, each up to 8 characters long.

❺ Variable names may be any combination of up to 32 numbers and alphabetic characters and the underscore character. They **must** start with a alphabetic character or an underscore.

❸ Variable names may be any combination of up to 8 numbers and alphabetic letters. They **must** start with a letter.

The length of a variable name includes the `%`, `&` or `$` sign, but not the `()` in string and array variables.

More than one GLOBAL or LOCAL statement may be used, but they must be on separate lines, immediately after the procedure name.

See also LOCAL.

GMOVE

Usage: `gMOVE dx%,dy%`

Moves the current position `dx%` to the right and `dy%` downwards, in the current drawable.

A negative `dx%` causes movement to the left; a negative `dy%` causes upward movement.

See also gAT.

GORDER

Usage: `gORDER id%,position%`

Sets the window specified by `id%` to the selected foreground/background position, and redraws the screen. Position 1 is the foreground window, position 2 is next, and so on. Any position greater than the number of windows is interpreted as the end of the list.

On creation, a window is at position 1 in the list.

Raises an error if `id%` is a bitmap.

See also gRANK.

GORIGINX

Usage: `x%=gORIGINX`

Returns the gap between the left side of the screen and the left side of the current window.

Raises an error if the current drawable is a bitmap.

GORIGINY

Usage: `y%=gORIGINY`

Returns the gap between the top of the screen and the top of the current window.

Raises an error if the current drawable is a bitmap.

GOTO

Usage: `GOTO label` *or* `GOTO label::`

```
       ...
       label::
```

Goes to the line following the `label::` and continues from there. The label

- Must be in the current procedure
- Must start with a letter and end with a double colon, although the double colon is not necessary in the GOTO statement
- May be up to 32 characters long on the Series 5, or 8 on the Series 3c, excluding the colons.

❺ GOTOMARK

Usage: `GOTOMARK b%`

Makes the record with bookmark `b%`, as returned by BOOKMARK, the current record. `b%` must be a bookmark in the current view.

GPATT

Usage: `gPATT id%,width%,height%,mode%`

Fills a rectangle of the specified size from the current position with repetitions of the drawable `id%`.

As with gCOPY, this command can copy both set and clear pixels, so the same modes are available as when displaying text. Set `mode%`=0 for set, 1 for clear, 2 for invert or 3 for replace. 0, 1 and 2 act only on set pixels in the pattern; 3 copies the entire rectangle, with set and clear pixels.

If you set `id%`=-1 a pre-defined grey pattern is used.

The current position is unaffected.

❸ gPATT is affected by the setting of gGREY (in the **current window**) in the same way as gCOPY: with `gGREY 0` it copies black to black; with `gGREY 1` it copies grey to grey, or black to grey if source is black only; with `gGREY 2` it copies grey to grey and black to black, or black to both if source is black only.

GPEEKLINE

Usage: `gPEEKLINE id%,x%,y%,d%(),ln%`

or ❺ `gPEEKLINE id%,x%,y%,d%(),ln%,mode%`

Reads a horizontal line from the black plane of the drawable `id%`, length `ln%`, starting at `x%,y%`. The leftmost 16 pixels are read into `d%(1)`, with the first pixel read into the least significant bit.

❸ If you set `id%` to 0, this just reads from the whole screen, not from any particular window.

❺ gPEEKLINE has an extra optional parameter `mode%` to specify the colour mode:

`mode%`	*colour mode*	*colour of pixel which sets bits*
-1	black and white	black
0	black and white	white
1	4-colour mode	white
2	16-colour mode	white

The default `mode%` is -1. For 4 and 16-colour modes, 2 and 4 bits per pixel respectively are used. This is to enable the colour of the pixel to be ascertained from the bits which are set. White results in all 2 or 4 bits being set, while black sets none of them. For example, in a 4-colour window, with the colour set by

```
gCOLOR 16,16,16
```

a pixel of a line would peek as `0001` in binary. Similarly, a pixel of a line with the colour set to

```
gCOLOR 80,80,80
```

would result in the value `0101` in binary when peeked.

The array `d%()` must be long enough to hold the data. You can work out the number of integers required with `((ln%+15)/16)` (using whole-number division).

❺ Note that if the optional parameter `mode%` is used on the Series 5, the array size allowed must be adjusted accordingly: it must be at least twice as long as the array needed for black and white if the line you wish to peek in 4-colour mode and four times as long in 16-colour mode.

On the Series 3c, if you add $8000 to `id%`, the grey plane (not the black plane) will be peeked, but note that $8000 ORed with the `id%` on the Series 5 will raise an ‘Invalid arguments’ error.

GPOLY

Usage: `gPOLY a%()`

Draws a sequence of lines, as if by gLINEBY and gMOVE commands.

The array is set up as follows:

`a%(1)` starting `x` position

`a%(2)` starting `y` position

`a%(3)` number of pairs of offsets

`a%(4)` `dx1%`

`a%(5)` `dy1%`

`a%(6)` `dx2%`

`a%(7)` `dy2%` etc.

❺ Constants for the first five array subscripts are supplied in Const.oph. See the ‘Calling Procedures’ csection 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.

Each pair of numbers `dx1%,dy1%`, for example specifies a line or a movement. To draw a line, `dy%` is the amount to move down, while `dx%` is the amount to move to the right **multiplied by two**.

To specify a movement (i.e. without drawing a line) work out the `dx%,dy%` as for a line, then add 1 to `dx%`.

(For drawing/movement up or left, use negative numbers.)

gPOLY is quicker than combinations of gAT, gLINEBY and gMOVE.

Example, to draw three horizontal lines 50 pixels long at positions 20,10, 20,30 and 20,50:

```
a%(1)=20 :a%(2)=10            REM 20,10
a%(3)=5                       REM 5 operations
a%(4)=50*2  :a%(5)=0          REM draw right 50
a%(6)=0*2+1 :a%(7)=20         REM move down 20
a%(8)=-50*2 :a%(9)=0          REM draw left 50
a%(10)=0*2+1 :a%(11)=20       REM draw left 50
a%(12)=50*2 :a%(13)=0         REM draw right 50
gPOLY a%()
```

GPRINT

Usage: `gPRINT` *`list of expressions`*

Displays a list of expressions at the current position in the current drawable. All variable types are formatted as for PRINT.

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 without a list of expressions does nothing.

See also gPRINTB, gPRINTCLIP, gTWIDTH, gXPRINT, gTMODE.

GPRINTB

Usage: any of

```
gPRINTB t$,w%,al%,tp%,bt%,m%
gPRINTB t$,w%,al%,tp%,bt%
gPRINTB t$,w%,al%,tp%
gPRINTB t$,w%,al%
gPRINTB t$,w%
```

Displays text `t$` in a cleared box of width `w%` pixels. The current position is used for the left side of the box and for the baseline of the text.

`al%` controls the alignment of the text in the box 1 for right aligned, 2 for left aligned, or 3 for centred.

`tp%` and `bt%` are the clearances between the text and the top/bottom of the box. Together with the current font size, they control the height of the box. An error is raised if `tp%` plus the font ascent is greater than 255.

`m%` controls the margins. For left alignment, `m%` is an offset from the left of the box to the start of the text. For right alignment, `m%` is an offset from the right of the box to the end of the text. For centring, `m%` is an offset from the left or right of the box to the region in which to centre, with positive `m%` meaning left and negative meaning right.

If values are not supplied for some arguments, these defaults are used:

`al%`	left
`tp%`	0
`bt%`	0
`m%`	0

❺ Constants for the layout features and the defaults 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.

See also gPRINT, gPRINTCLIP, gTWIDTH, gXPRINT.

GPRINTCLIP

Usage: `w%=gPRINTCLIP(text$,width%)`

Displays `text$` at the current position, displaying only as many characters as will fit inside `width%` pixels. Returns the number of characters displayed.

See also gPRINT, gPRINTB, gTWIDTH, gXPRINT, gTMODE.

GRANK

Usage: `rank%=gRANK`

Returns the foreground/background position, from 1 to 64 on the Series 5, and 1 to 8 on the Series 3c, of the current window.

Raises an error if the current drawable is a bitmap.

See also gORDER.

GSAVEBIT

Usage: `gSAVEBIT name$,width%,height%`

or `gSAVEBIT name$`

Saves the current drawable as the named bitmap file. If `width%` and `height%` are given, then only the rectangle of that size from the current position is copied.

gSAVEBIT does not add a default filename extension to the input argument name if none is provided on the Series 5, while on the Series 3c, if `name$` has no file extension `.PIC` is used.

❸ Saving a window to file when it includes grey will save both planes to the file, black bitmap first followed by grey.

GSCROLL

Usage: `gSCROLL dx%,dy%,x%,y%,wd%,ht%`

or `gSCROLL dx%,dy%`

Scrolls pixels in the current drawable by offset `dx%,dy%`. Positive `dx%` means to the right, and positive `dy%` means down. The drawable itself does not change its position.

If you specify a rectangle in the current drawable, at `x%,y%` and of size `wd%,ht%`, only this rectangle is scrolled.

The areas `dx%` wide and `dy%` deep which are “left behind” by the scroll are cleared.

The current position is not affected.

❺ GSETPENWIDTH

Usage: `gSETPENWIDTH width%`

Sets the pen width in the current drawable to `width%` pixels.

GSETWIN

Usage: `gSETWIN x%,y%,width%,height%`

or `gSETWIN x%,y%`

Changes position and, optionally, the size of the current window.

An error is raised if the current drawable is a bitmap.

The current position is unaffected.

If you use this command on the default window, you must also use the SCREEN command to ensure that the area for PRINT commands to use is wholly contained within the default window.

GSTYLE

Usage: `gSTYLE style%`

Sets the style of text displayed in subsequent gPRINT, gPRINTB and gPRINTCLIP commands on the current drawable.

`style%`	*text style*
0	normal
1	bold
2	underlined
4	inverse
8	double height
16	mono-spaced
32	italic

You can combine these styles by adding their values for example, to set bold, underlined and double height, use `gSTYLE 11`, as 11=1+2+8.

This command does not affect non-graphics commands, like PRINT.

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 Appendix E in the ‘Appends.pdf’ document for a listing of it.

GTMODE

Usage: `gTMODE mode%`

Sets the way characters are displayed by subsequent gPRINT and gPRINTCLIP commands on the current drawable.

`mode%`	*pixels will be*
0	set
1	cleared
2	inverted
3	replaced

When you first use graphics text commands on a drawable, each dot in a letter causes a pixel to be set in the drawable. This is `mode%`=0.

When `mode%` is 1 or 2, graphics text commands work in a similar way, but the pixels are cleared or inverted. When `mode%` is 3, entire character boxes are drawn on the screen - pixels are set in the letter and cleared in the background box.

This command does not affect other text display commands.

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 Appendix E in the ‘Appends.pdf’ document for a listing of it.

GTWIDTH

Usage: `width%=gTWIDTH(text$)`

Returns the width of `text$` in the current font and style.

See also gPRINT, gPRINTB, gPRINTCLIP, gXPRINT.

GUNLOADFONT

Usage: ❺ `gULOADFONT fileId%`

❸ `gUNLOADFONT fontId%`

Unloads a user-defined font that was previously loaded using gLOADFONT. Raises an error if the font has not been loaded.

The built-in fonts are not held in memory and cannot be unloaded.

See also gLOADFONT.

GUPDATE

Usage: any of

```
gUPDATE ON
gUPDATE OFF
gUPDATE
```

The Psion's screen is usually updated whenever you display anything on it. `gUPDATE OFF` switches off this feature. The screen will then be updated as few times as possible (though note that some keywords will always cause an update.) You can still force an update by using the gUPDATE command on its own.

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.

gUPDATE affects anything that displays on the screen. If you are using a lot of PRINT commands, `gUPDATE OFF` may make a noticeable difference in speed.

Note that with gUPDATE OFF, the location of errors which occur while the procedure is running may be incorrectly reported. For this reason, gUPDATE OFF is best used in the final stages of program development, and even then you may have to remove it to locate some errors.

GUSE

Usage: `gUSE id%`

Makes the drawable `id%` current. Graphics drawing commands will now go to this drawable. gUSE does not bring a drawable to the foreground (see gORDER).

GVISIBLE

Usage: `gVISIBLE ON`

or `gVISIBLE OFF`

Makes the current window visible or invisible.

Raises an error if the current drawable is a bitmap.

GWIDTH

Usage: `width%=gWIDTH`

Returns the width of the current drawable.

GX

Usage: `x%=gX`

Returns the `x` current position (in from the left) in the current drawable.

GXBORDER

Usage: `gXBORDER type%,flags%,w%,h%`

or `gXBORDER type%,flags%`

Draws a border in the current drawable of a specified type, fitting inside a rectangle of the specified size or with the size of the current drawable if no size is specified.

`type%=1` for drawing the Series 3c 3-D grey and black border. A shadow or a gap for a shadow is always assumed.

`type%=2` for drawing the Series 5 borders.

❺ Values for `flags%` and their effects on the Series 5 are as follows,

border type	`flags%`
none	$00
single black	$01
shallow sunken	$42
deep sunken	$44
deep sunken with outline	$54
shallow raised	$82
deep raised	$84
deep raised with outline	$94
vertical bar	$22
horizontal bar	$2a

Constants for these flags and types 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.

❸ On the Series 3c, `flags%=1,2,3,4` are as for gBORDER. When the shadow is enabled (1 or 3) only the grey and black parts of the border are drawn; you should pre-clear the background for the white parts. When the shadow is disabled (2 or 4) the outer and inner border lines are drawn, but the areas covered by grey/black when the shadow is enabled are now cleared. (This allows a shadow to be turned off simply by calling gXBORDER again.)

On the Series 3c, an error is raised if the current window has no grey plane.

The following values of `flags%` apply to all border types:

0 for normal corners

Adding $100 leaves 1 pixel gap around the border.

Adding $200 for more rounded corners

Adding $400 for losing a single pixel.

If both $400 and $200 are mistakenly supplied, $200 has priority.

See also gBORDER.

GXPRINT

Usage: `gXPRINT string$,flags%`

Displays `string$` at the current position, with precise highlighting or underlining. The current font and style are still used, even if the style itself is inverse or underlined. If text mode 3 (replace) is used both set and cleared pixels in the text are drawn.

`flags%` has the following effect:

`flags%`	*effect*
0	normal, as with gPRINT
1	inverse
2	inverse, except corner pixels
3	thin inverse
4	thin inverse, except corner pixels
5	underlined
6	thin underlined

❺ 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.

Where lines of text are separated by a single pixel, the **thin** options maintain the separation between lines.

gXPRINT does not support the display of a list of expressions of various types.

GY

Usage: `y%=gY`

Returns the `y` current position (down from the top) in the current drawable.

HEX$

Usage: `h$=HEX$(x&)`

Returns a string containing the hexadecimal (base 16) representation of integer or long integer `x&`.

For example `HEX$(255)` returns the string `“FF”`.

NOTES

To enter integer hexadecimal constants (16 bit) put a `$` in front of them. For example `$FF` is 255 in decimal. (Don’t confuse this use of `$` with string variable names.)

To enter long integer hexadecimal constants (32 bit) put a `&` in front of them. For example `&FFFFF` is 1048575 in decimal.

Counting in hexadecimal is done like this: `0 1 2 3 4 5 6 7 8 9 A B C D E F 10...` where `A` stands for decimal 10, `B` for decimal 11, `C` for decimal 12 ... up to `F` for decimal 15. After `F` comes `10`, which is equivalent to decimal 16. To understand numbers greater than hexadecimal 10, again compare hexadecimals with decimals. In these examples, 10^2 means 10×10, 10^3 means 10×10×10 and so on.

253 in decimal is:

$(2\times10^2)+(5\times10^1)+(3\times10^0) = (2\times100)+(5\times10)+(3\times1) = 200+50+3$

By analogy, &253 in hexadecimal is:

$(\&2\times16^2)+(\&5\times16^1)+(\&3\times16^0) =(2\times256)+(5\times16)+(3\times1) =512+80+3 = 595$ in decimal.

Similarly, &A6B in hexadecimal is:

$(\&A\times16^2)+(\&6\times16^1)+(\&B\times16^0) =(10\times256)+(6\times16)+(11\times1) =2560+96+11 = 2667$ in decimal.

You may also find this table useful for converting between hex and decimal:

hex	*decimal*	
&1	1	$=16^0$
&10	16	$=16^1$
&100	256	$=16^2$
&1000	4096	$=16^3$

For example, &20F9 is

$(2\times\&1000)+(0\times\&100)+(15\times\&10)+9$ which in decimal is: $(2\times4096)+(0\times256)+(15\times16)+9 = 8441$.

All hexadecimal constants are integers ($) or long integers (&). So arithmetic operations involving hexadecimal numbers behave in the usual way. For example, `&3/&2` returns `1`, `&3/2.0` returns `1.5`, `3/$2` returns `1`.

HOUR

Usage: `h%=HOUR`

Returns the number of the current hour from the system clock as an integer between 0 and 23.

IABS

Usage: `i&=IABS(x&)`

Returns the absolute value, i.e. without any sign, of the integer or long integer expression `x&`.

For example `IABS(-10)` is `10`.

See also ABS, which returns the absolute value as a floating-point value.

ICON

Usage: `ICON mbm$`

Gives the name of the bitmap file `mbm$` (also known as an EPOC Picture file) to use as the icon for an OPL Application.

If the ICON command is not used inside the APP…ENDA structure, then a default icon is used, but the rest of the information in the APP..ENDA construct is still used to specify the other features of the OPL application.

❺ On the Series 5, `mbm$` is a multi-bitmap file which can contain up to three bitmap/mask pairs - the sizes are 24, 32 and 48 squares. These different sizes are used for the different zoom levels in the system screen. The sizes are read from the MBM and the most suitable size is zoomed if the exact sizes required are not provided or if some are missing.

In fact, you can use ICON more than once within the APP...ENDA construct on the Series 5. The translator only insists that all icons are paired with a mask of the same size in the final ICON list. This allows you to use pairs of MBMs containing just one bitmap as produced by the Sketch application. For example, you could specify them individually:

```
APP ...
 ICON “icon24.mbm”
 ICON “mask24.mbm”
 ICON “icon32.mbm”
 ICON “mask32.mbm”
 ICON “icon48.mbm”
 ICON “mask48.mbm”
ENDA
```

or with pairs in each MBM:

```
APP ...
 ICON “iconMask24”
 ICON “iconMask32”
 ICON “iconMask48”
ENDA
```

or with all the bitmaps as just one MBM, as would normally be the case if prepared on the PC using the BMCONV tool and the AIFTOOL (description of these tools is beyond the scope of this manual and you should refer to the EPOC32 C++ Software Development Kit (SDK), which is available from Psion Software plc, for more details).

This command can only be used between APP and ENDA.

See the ‘Advanced.pdf’ document for more details of OPL applications.

IF...ENDIF

Usage:
```
IF condition1
    ...
ELSEIF condition2
    ...
ELSE
    ...
ENDIF
```

Does **either**

- the statements following the IF condition

or

- the statements following one of the ELSEIF conditions (there may be as many ELSEIF statements as you like none at all if you want)

or

- the statements following ELSE (or, if there is no ELSE, nothing at all). There may be either one ELSE statement or none.

After the ENDIF statement, the lines following ENDIF carry on as normal.

IF, ELSEIF, ELSE and ENDIF **must** be in that order.

Every IF **must** be matched with a closing ENDIF.

You can also have an IF...ENDIF structure within another, for example:

```
IF condition1
   ...
ELSE
   ...
   IF condition2
      ....
   ENDIF
   ...
ENDIF
```

`condition` is an expression returning a logical value for example `a<b`. If the expression returns logical true (non-zero) then the statements following are executed. If the expression returns logical false (zero) then those statements are ignored. For more details about logical expressions, see Appendix B.

❺ INCLUDE

Usage: `INCLUDE file$`

Includes a file, `file$`, which may contain CONST definitions, prototypes for OPX procedures and prototypes for module procedures. The included file may **not** include module procedures themselves. Procedure and OPX procedure prototypes allow the translator to check parameters and coerce numeric parameters (that are not passed by reference) to the required type.

Including a file is logically identical to replacing the INCLUDE statement with the file’s contents.

The filename of the header may or may not include a path. If it does include a path, then OPL will only scan the specified folder for the file. However, the default path for INLCUDE is `\System\Opl\`, so when INCLUDE is called without specifying a path, OPL looks for the file firstly in the current folder and then in `\System\Opl\` in all drives from `Y:` to `A:` and then in `Z:`, excluding any remote drives.

See CONST, EXTERNAL. See also the ‘OPX.pdf’ document.

INPUT

Usage: `INPUT variable`

or `INPUT log.field`

Waits for a value to be entered at the keyboard, and then assigns the value entered to a variable or data file field.

You can edit the value as you type it in. All the usual editing keys are available: the arrow keys move along the line, Esc clears the line and so on.

If inappropriate input is entered, for example a string when the input was to be assigned to an integer variable, a `?` is displayed and you can try again. However, if you used `TRAP INPUT`, control passes on to the next line of the procedure, with the appropriate error condition being set and the value of the variable remaining unchanged.

INPUT is usually used in conjunction with a PRINT statement:

```
PROC exch:
   LOCAL pds,rate
   DO
      PRINT “Pounds Sterling?”,
      INPUT pds
      PRINT “Rate (DM)?”,
      INPUT rate
      PRINT “=“,pds*rate,”DM”
      GET
   UNTIL 0
ENDP
```

Note the commas at the end of the PRINT statements, used so that the cursor waiting for input appears on the same line as the messages.

TRAP INPUT

If a bad value is entered (for example `“abc”` for `a%`) in response to a TRAP INPUT, the `?` is not displayed, but the ERR function can be called to return the value of the error which has occurred. If the Esc key is pressed while no text is on the input line, the ‘Escape key pressed’ error (number -114) will be returned by ERR (provided that the INPUT has been trapped). You can use this feature to enable someone to press the Esc key to escape from inputting a value.

See also EDIT. This works like INPUT, except that it displays a string to be edited and then assigned to a variable or field. It can only be used with strings.

❺ INSERT

Usage: `INSERT`

Inserts a new, blank record into the current view of a database. The fields can then be assigned to before using PUT or CANCEL.

INT

Usage: `i&=INT(x)`

Returns the integer (in other words the whole number) part of the floating-point expression `x`. The number is returned as a long integer.

Positive numbers are rounded down, and negative numbers are rounded up for example `INT(-5.9)` returns `-5` and `INT(2.9)` returns `2`. If you want to round a number to the nearest integer, add 0.5 to it (or subtract 0.5 if it is negative) before you use INT.

❸ In the Calculator, you need to use INT to pass a number to a procedure which requires a long integer parameter. This is because the Calculator passes all numbers as floating-point by default.

See also INTF.

INTF

Usage: `i=INTF(x)`

Used in the same way as the INT function, but the value returned is a floating-point number. For example, `INTF(1234567890123.4)` returns `1234567890123.0`

You may also need this when an integer calculation may exceed integer range.

See also INT.

❺ INTRANS

Usage: `i&=INTRANS`

Finds out whether the current view is in a transaction. Returns -1 if it is in a transaction or 0 if it is not.

See also BEGINTRANS.

I/O FUNCTIONS

These functions are covered in detail in the ‘Advanced.pdf’ document.

`r%=IOA(h%,f%,var status%,var a1,var a2)`

This has the same form as IOC, but it returns an error value of the request is not completed successfully. IOC should be used in preference to IOA.

`IOC(h%,f%,var status%,var a1,var a2)`

Make an I/O request with guaranteed completion. The device driver opened with handle `h%` performs the asynchronous I/O function `f%` with two further arguments, `a1` and `a2`. The argument `status%` is set by the device driver. If an error occurs while making a request, `status%` is set to an appropriate values, but IOC always returns zero, not an error value. An IOWAIT or IOWAITSTAT must be performed for each IOC. IOC should be used in preference to IOA.

`r%=IOCANCEL(h%)`

Cancels any outstanding asynchronous I/O request (IOC or IOA). Note, however, that the request will still complete, so the signal must be consumed using IOWAITSTAT.

`r%=IOCLOSE(h%)`

Closes a file with the handle `h%`.

`r%=IOOPEN(var h%,name$,mode%)`

Creates or opens a file called `name$`. Defines `h%` for use by other I/O functions. `mode%` specifies how to open the file. For unique file creation, use `IOOPEN(var h%,addr%,mode%)`

❺ `r%=IOREAD(h%,addr&,maxLen%)`

❸ `r%=IOREAD(h%,addr%,maxLen%)`

Reads from the file with the handle `h%`. `address%` is the address of a buffer large enough to hold a maximum of `maxLen%` bytes. The value returned to `r%` is the actual number of bytes read or, if negative, is an error value.

`r%=IOSEEK(h%,mode%,var off&)`

Seeks to a position in a file that has been opened for random access. `mode%` specifies how the offset argument `off&` is to be used. Values for `mode%` may be found in the ‘Advanced.pdf’ document. `off&` may be positive to move forwards or negative to move backwards. IOSEEK sets the variable `off&` to the absolute position set.

`IOSIGNAL`

Signals an asynchronous I/O function’s completion.

`r%=IOW(h%,func%,var a1,var a2)`

The device driver opened with handle `h%` performs the synchronous I/O function `func%` with the two further arguments.

`IOWAIT`

Waits for an asynchronous I/O function to signal completion.

`IOWAITSTAT var stat%`

Waits for an asynchronous function, called with IOC or IOA, to complete.

❺ `IOWAITSTAT32 var stat&`

Takes a 32-bit status word. IOWAITSTAT32 should be called only when you need to wait for completion of a request made using a 32-bit status word when calling an asynchronous OPX procedure.

✎ The initial value of a 32-bit status word while it is still pending (i.e. waiting to complete) is &80000001 (`KStatusPending32&` in Const.oph: see the ‘Calling Procedures’ section of the ‘Basics.pdf’ document for details of how to use this file). For a 16-bit status word the ‘pending value’ is -46 (`KErrFilePending%`).

❺ `r%=IOWRITE(h%,addr&,length%)`

❸ `r%=IOWRITE(h%,addr%,length%)`

Writes `length%` bytes in a buffer at `address%` to the file with the handle `h%`.

`IOYIELD`

Ensures that any asynchronous handler set up with IOC or IOA is given a chance to run. IOYIELD must always be called before polling status words on the Series 5, i.e. before reading a 16-bit status word if IOWAIT or IOWAITSTAT have not been used first.

Note the following example when you use #:

On the Series 3c you would call IOSEEK using,

```
ret%=IOSEEK(h%,mode%,#ptrOff%)
```

but on the Series 5 you would use,

```
ret%=IOSEEK(h%,mode%,#ptrOff&)
```

passing the long integer `ptrOff&`.

See also ‘32-bit addressing’ in the ‘Advanced.pdf’ document.

KEY

Usage: `k%=KEY`

Returns the character code of the key last pressed, if there has been a keypress since the last use of the keyboard by INPUT, EDIT, GET, GET$, KEY, KEY$, MENU and DIALOG.

If no key has been pressed, zero is returned.

See Appendix D in the 'Appends.pdf' document for a list of special key codes. You can use KMOD to check whether *modifier keys* (Shift, Control, Fn (on the Series 5), Psion (on the Series 3c) and Caps Lock) were used.

This command does not wait for a key to be pressed, unlike GET.

KEY$

Usage: `k$=KEY$`

Returns the last key pressed as a string, if there has been a keypress since the last use of the keyboard by INPUT, EDIT, GET, GET$, KEY, KEY$, MENU and DIALOG.

If no key has been pressed, a null string (`""`) is returned.

See Appendix D in the 'Appends.pdf' document for a list of special key codes. You can use KMOD to check whether *modifier keys* (Shift, Control, Fn (on the Series 5), Psion (on the Series 3c) and Caps Lock) were used.

This command does not wait for a key to be pressed, unlike GET$.

KEYA

Usage: `err%=KEYA(var stat%,var key%(1))`

This is an asynchronous keyboard read function.

See the 'Advanced.pdf' document for details.

Cancel with KEYC.

KEYC

Usage: `err%=KEYC(var stat%)`

Cancels the previously called KEYA function with status `stat%`. Note that KEYC consumes the signal (unlike IOCANCEL), so IOWAITSTAT should not be used after KEYC.

See the 'Advanced Topics' section of the 'Advanced' document for details.

❺ KILLMARK

Usage: `KILLMARK b%`

Removes the bookmark `b%`, which has previously been returned by BOOKMARK, from the current view of a database.

See BOOKMARK, GOTOMARK.

KMOD

Usage: `k%=KMOD`

Returns a code representing the state of the modifier keys (whether they were pressed or not) at the time of the last keyboard access, such as a KEY function. The modifiers have these codes:

	decimal code		*hex code*
	2	Shift down	$2
	4	Control down	$4
❸	8	Psion down	$6
	16	Caps Lock on	$10
❺	128	Fn down	$80

❺ Constants for these codes 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.

If there was no modifier, the function returns 0. If a combination of modifiers was pressed, the *sum* of their codes is returned - for example 20 is returned if Control (4) was held down and Caps Lock (16) was on.

Always use immediately after a KEY/KEY$/GET/GET$ statement.

The value returned by KMOD has one binary bit set for each modifier, as shown above. By using the logical operator AND on the value returned by KMOD you can check which of the bits are set, in order to see which modifier keys were held down. For more details on AND, see Appendix B in the ‘Appends.pdf’ document.

Example:

```
PROC modifier:
  LOCAL k%,mod%
  PRINT “Press a key” :k%=GET
  CLS :mod%=KMOD
  PRINT “Key code”,k%,”with”
  IF mod%=0
    PRINT “no modifier”
  ENDIF
  IF mod% AND 2
    PRINT “Shift down”
  ENDIF
  IF mod% AND 4
    PRINT “Control down”
  ENDIF
  IF mod% AND 8                  REM only needed on Series 3c
    PRINT “Psion down”
  ENDIF
  IF mod% AND 16
    PRINT “Caps Lock on”
  ENDIF
  IF mod% AND 128                REM only needed on Series 5
    PRINT “Fn down”
  ENDIF
ENDP
```

LAST

Usage: `LAST`

Positions to the last record in a data file.

LCLOSE

Usage: `LCLOSE`

Closes the device opened with LOPEN. (The device is also closed automatically when a program ends.)

LEFT$

Usage: `b$=LEFT$(a$,x%)`

Returns the leftmost `x%` characters from the string `a$`.

For example if `n$` has the value `Charles`, then `b$=LEFT$(n$,3)` assigns `Cha` to `b$`.

LEN

Usage: `a%=LEN(a$)`

Returns the number of characters in `a$`.

E.g. if `a$` has the value `34 Kopechnie Drive` then `LEN(a$)` returns `18`.

You might use this function to check that a data file string field is not empty before displaying:

```
IF LEN(A.client$)
  PRINT A.client$
ENDIF
```

LENALLOC

Usage: ❺ `len&=LENALLOC(pcell&)`

❸ `len%=LENALLOC(pcell%)`

Returns the length of the previously allocated cell at `pcell&` (`pcell%` on the Series 3c).

❺ Cells are allocated lengths that are the smallest multiple of four greater than the size originally requested. An error will be raised if the cell address argument is not in the range known by the heap.

See also SETFLAGS if you require the 64K limit to be enforced on the Series 5. If the flag is set to restrict the limit, `len&` is guaranteed to fit into an integer.

❸ LINKLIB

Usage: `LINKLIB cat%`

Link any libraries that have been loaded using LOADLIB.

❺ The Series 5 supports use of OPXs only. See the ‘OPX.pdf’ document for further details.

LN

Usage: `a=LN(x)`

Returns the natural (base e) logarithm of `x`.

Use LOG to return the base 10 log of a number.

❸ LOADLIB

Usage: `ret%=LOADLIB(var cat%,name$,link%)`

Load and optionally link a DYL that is not in the ROM. If successful, this writes the category handle to `cat%` and returns zero. The DYL is shared in memory if already loaded by another process.

❺ The Series 5 supports use of OPXs only. See the ‘OPX.pdf’ document for further details.

LOADM

Usage: `LOADM module$`

Loads a translated OPL module so that procedures in that module can be called. Until a module is loaded with LOADM, calls to procedures in that module will give an error.

`module$` is a string containing the name of the module. Specify the full file name only where necessary.

Example: `LOADM “MODUL2”`

Up to 8 modules on the Series 5, or 4 on the Series 3c, can be in memory at any one time, including the top level module; if you try to LOADM a ninth (fifth) module, you get an error. Use UNLOADM to remove a module from memory so that you can load a different one.

❺ By default, LOADM always uses the folder of the top level module. It is not affected by the SETPATH command.

❸ By default, LOADM always uses the directory of the initial running program, or the one specified by a OPA application. It is not affected by the SETPATH command.

LOC

Usage: `a%=LOC(a$,b$)`

Returns an integer showing the position in `a$` where `b$` occurs, or zero if `b$` doesn’t occur in `a$`. The search matches upper and lower case.

Example: `LOC(“STANDING”,”AND”)` would return the value `3` because the substring `AND` starts at the third character of the string `STANDING`.

LOCAL

Usage: `LOCAL` *`variables`*

Used to declare variables which can be referenced only in the current procedure. Other procedures may use the same variable names to create new variables. Use GLOBAL to declare variables common to all called procedures.

The variables may be of 4 types, depending on the symbol they end with:

- Variable names not ending with `$`, `%`, `&` or `()` are floating-point variables, for example price, `x`
- Those ending with a `%` are integer variables, for example `x%`, `sales92%`
- Those ending with an `&` are long integer variables, for example `x&`, `sales92&`.
- Those ending with a `$` are string variables. String variable names must be followed by the maximum length of the string in brackets, for example `names$(12)`, `a$(3)`

Array variables have a number immediately following them in brackets which specifies the number of elements in the array. Array variables may be of any type, for example: `x(6),y%(5),f$(5,12),z&(3)`

When declaring string arrays, you must give two numbers in the brackets. The first declares the number of elements, the second declares their maximum length. For example `surname$(5,8)` declares five elements, each up to 8 characters long.

❺ Variable names may be any combination of up to 32 numbers, alphabetic letters and the underscore character. They **must** start with a letter or an underscore. The length includes the `%`, `&` or `$` sign, but not the `()` in string and array variables.

❸ Variable names may be any combination of up to 8 numbers and alphabetic letters. They **must** start with a letter. The length includes the `%`, `&` or `$` sign, but not the `()` in string and array variables.

More than one GLOBAL or LOCAL statement may be used, but they **must** be on separate lines, immediately after the procedure name.

See also GLOBAL, CONST and the ‘Variables and Constants’ section of the ‘Basics.pdf’ document.

LOCK

Usage: `LOCK ON`

or `LOCK OFF`

Mark an OPA (OPL application) as locked or unlocked. When an OPA is locked with `LOCK ON`, the System screen will not send it events to change files or quit.

❺ If, for example, you move to the task list or the document name in the System screen try to stop that running OPA by using the ‘Close file’ button or Ctrl+E respectively, a message will appear, indicating that the OPA cannot close down at that moment.

❸ If, for example, you move on to the file list in the System screen and press Delete to try to stop that running OPA, a message will appear, indicating that the OPA cannot close down at that moment.

You should use `LOCK ON` if your OPA uses a command, such as EDIT, MENU or DIALOG, which accesses the keyboard. You might also use it when the OPA is about to go busy for a considerable length of time, or at any other point where a clean exit is not possible. Do not forget to use `LOCK OFF` as soon as possible afterwards.

An OPA is initially unlocked.

LOG

Usage: `a=LOG(x)`

Returns the base 10 logarithm of `x`.

Use LN to find the base e (natural) log.

LOPEN

Usage: `LOPEN device$`

Opens the device to which LPRINTs are to be sent.

No LPRINTs can be sent until a device has been opened with LOPEN.

You can open any of these devices:

- ❸ The parallel port, with `LOPEN “PAR:A”`
- The serial port, with `LOPEN “TTY:A”`
- A file on the Psion
- ❸ A file on an attached computer. LOPEN the file name, e.g. on a PC:

  `LOPEN “REM::C:\BAK\MEMO.TXT”`

  or on an Apple Macintosh:

  `LOPEN “REM::HD40:ME:MEMO5”`

Any existing file of the name given will be overwritten when you print to it.

Only one device may be open at any one time. Use LCLOSE to close the device. (It also closes automatically when a program finishes running.)

See Appendix C.

LOWER$

Usage: `b$=LOWER$(a$)`

Converts any upper case characters in the string `a$` to lower case and returns the completely lower case string.

E.g. if `a$=“CLARKE”`, `LOWER$(a$)` returns the string `“clarke”`

Use UPPER$ to convert a string to upper case.

LPRINT

Usage: `LPRINT` *`list of expressions`*

Prints a list of items, in the same way as PRINT, except that the data is sent to the device most recently opened with LOPEN.

The expressions may be quoted strings, variables, or the evaluated results of expressions. The punctuation of the LPRINT statement (commas, semicolons and new lines) determines the layout of the printed text, in the same way as PRINT statements.

If no device has been opened with LOPEN you will get an error.

See PRINT for displaying to the screen.

See LOPEN for opening a device for LPRINT.

See Appendix C in the ‘Appends.pdf’ document for an overview of printing from OPL. See also Printer OPX in the ‘OPX.pdf’ document for details of more advanced printing features.

MAX

Usage: `m=MAX(list)`

or `m=MAX(array(),element)`

Returns the greatest of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on for example `m=MAX(arr(),3)` would return the value of the largest of elements `arr(1)`, `arr(2)` and `arr(3)`.

MCARD

Usage: `mCARD title$,n1$,k1%`

or `mCARD title$,n1$,k1%,n2$,k2%` etc.

Defines a menu. When you have defined all of the menus, use MENU to display them.

`title$` is the name of the menu. From one to eight items on the menu may be defined, each specified by two arguments. The first is the item name, and the second the keycode for a shortcut key. This specifies a key which, when pressed together with the Ctrl (Psion on the Series 3c) key, will select the option. If the keycode is for an upper case key, the shortcut key will use both the Shift and Ctrl (Psion) keys.

The options can be divided into logical groups by displaying a separating line under the final option in a group. To do this, pass the negative value corresponding to the shortcut key keycode for the final option in the group. For example, `-%A` specifies shortcut key Ctrl+Shift+A (Shift-Psion-A in Series 3c) and displays a separating line under the associated option in the menu.

❺ Series 5 supports the following menu features

- Menu items without shortcuts, by specifying shortcut values between 1 and 32. For these items the value specified is still returned if the item is selected.
- Menu items which are dimmed, or which have checkboxes or option buttons (sometimes known as radio buttons).

These extra properties are controlled by adding the following bits to the shortcut key keycode.

effect	*value*
menu item dimmed	$1000
item has check-box	$0800
start of an option button list	$0900
middle of an option button list	$0A00
end of an option button list	$0B00
checkbox/option button symbol on	$2000
checkbox/option button symbol indeterminate	$4000

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.

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. 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.

If a separating line is required when any of these effects had been added, you must be sure to negate the whole value, not just the shortcut key keycode. In the example,

```
mCARD “Options”,“View1”,%A OR $2900,“View2”,-(%B OR $B00), “Another
      option”,%C
```

the second shortcut key keycode and its flag value is correctly negated to display a separating line.

A ‘Too wide’ error is raised if the menu title length is greater than or equal to 40. 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.

If any menu item fails to be added successfully, a menu is discarded. It is therefore incorrect to ignore mCARD errors by having an ONERR label around an mCARD call on the Series 5. If you do, the menu is discarded and a ‘Structure fault’ will be raised on using mCARD without first using mINIT again. See MENU for an example of this.

See mCASC for cascaded menu items.

See also the ‘Friendlier Interaction’ section of the ‘GUI.pdf’ document.

➎ MCASC

Usage: `mCASC title$,item1$,hotkey1%,item2$,hotkey2%`

Creates a cascade for a menu, on which less important menu items can be displayed. The cascade must be defined before use in a menu card. For example, a ‘Bitmap’ cascade under the File menu of a possible OPL drawing application could be defined like this:

```
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. To display a > in a cascaded menu item, you can use >>.

Shortcut keys used in cascades may be added to the appropriate constant values as for mCARD 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.

See mCARD, MENU, mINIT.

MEAN

Usage: `m=MEAN(list)`

or `m=MEAN(array(),element)`

Returns the arithmetic mean (average) of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on for example `m=MEAN(arr(),3)` would return the average of elements `arr(1)`, `arr(2)` and `arr(3)`.

This example displays `15.0`:

```
a(1)=10

a(2)=15

a(3)=20

PRINT MEAN(a(),3)
```

MENU

Usage: `val%=MENU`

or `val%=MENU(var init%)`

Displays the menus defined by mINIT, mCARD and mCASC, and waits for you to select an item. Returns the shortcut key keycode of the item selected, as defined in mCARD, **in lower case**.

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

If the name of a variable is passed, it sets the initial menu pane and item to be highlighted. `init%` should be `256*(menu%)+item%`; for both `menu%` and `item%`, 0 specifies the first, 1 the second and so on. If `init%` is 517 (=256*2+5), for example, this specifies the 6th item on the third menu.

If `init%` was passed, MENU writes back to `init%` the value for the item which was last highlighted on the menu. You can then use this value when calling the menu again.

❸ You only need to use this technique if you have more than one menu in your program, maintaining one initialisation variable for each.

When the menu is closed and reopened the highlighted item on reopening is the one last selected.

❺ It is necessary to use `MENU(init%)`, passing back the same variable each time the menu is opened if you wish the menu to reopen with the highlight set on the last selected item.

On the Series 5, it is incorrect to ignore mCARD and mCASC errors by having an ONERR label around an mCARD or mCASC call. If you do, the menu is discarded and a ‘Structure fault’ will be raised on using mCARD, mCASC or MENU without first using mINIT again.

The following bad code will not display the menu:

```
         mINIT
         ONERR errIgnore1
         mCARD “Xxx”,”ItemA”,0   REM bad shortcut
errIgnore1::
         ONERR errIgnore2
         mCARD “Yyy”,””       REM ‘Structure fault’ error (mINIT discarded)
errIgnore2::
         ONERR OFF
         MENU                 REM ‘Structure fault’ again
```

MID$

Usage: `m$=MID$(a$,x%,y%)`

Returns a string comprising `y%` characters of `a$`, starting at the character at position `x%`.

E.g. if `name$=“McConnell”` then `MID$(name$,3,4)` would return the string `Conn`.

MIN

Usage: `m=MIN(list)`

or `m=MIN(array(),element)`

Returns the smallest of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on for example `m=MIN(arr(),3)` would return the minimum of elements `arr(1)`, `arr(2)` and `arr(3)`.

MINIT

Usage: `mINIT`

Prepares for definition of menus, cancelling any existing menus. Use mCARD and mCASC (on the Series 5 only) to define each menu, then MENU to display them.

❺ On the Series 5, it is incorrect to ignore mCARD or mCASC errors by having an ONERR label around an mCARD or mCASC call. If you do, the menu is discarded and a ‘Structure fault’ will be raised if there is an occurrence of mCARD, mCASC or MENU without first using mINIT again. See also MENU for an example of this.

MINUTE

Usage: `m%=MINUTE`

Returns the current minute number from the system clock (0 to 59).

E.g. at 8.54am `MINUTE` returns `54`.

MKDIR

Usage: `MKDIR name$`

Creates a new folder/directory.

❺ For example, `MKDIR “C:\MINE\TEMP”` creates a `C:\MINE\TEMP` folder, also creating `C:\MINE` if it is not already there.

❸ For example, `MKDIR “M:\MINE\TEMP”` creates a `M:\MINE\TEMP` directory, also creating `M:\MINE` if it is not already there.

❺ MODIFY

Usage: `MODIFY`

Allows the current record of a view to be modified without moving the record. The fields can then be assigned to before using PUT or CANCEL.

MONTH

Usage: `m%=MONTH`

Returns the current month from the system clock as an integer between 1 and 12.

E.g. on 12th March 1992 `m%=MONTH` returns `3` to `m%`.

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

MONTH$

Usage: `m$=MONTH$(x%)`

Converts `x%`, a number from 1 to 12, to the month name, expressed as a three-letter mixed case string.

E.g. `MONTH$(1)` returns the string `Jan`.

❺ Constants for the month numbers are given in Const.oph. For details of how to use this file, see the “Calling Procedures” section of the ‘Basics.pdf’ document and for a lisitng of it see Appendix E in the ‘Appends.pdf’ document.

❺ MPOPUP

Usage: `mPOPUP(x%,y%,posType%,item1$,key1%,item2$,key2%,...)`

Presents a popup menu. mPOPUP returns the value of the keypress used to exit the popup menu, this being 0 if Esc is pressed.

⚠ Note that mPOPUP defines and presents the menu itself, and **should not** and **need not** be called from inside the mINIT…MENU structure.

`posType%` is the position type controlling which corner of the popup menu `x%,y%` specifies and can take the values,

`posType%`	*corner*
0	top left
1	top right
2	bottom left
3	bottom right

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.

`item$` and `key%` can take the same values as for mCARD, with `key%` taking the same constant values to specify checkboxes, option buttons and dimmed items. However, cascades in popup menus are not supported.

For example

```
mPOPUP (0,0,0,“Continue”,%c,“Exit”,%e)
```

specifies a popup menu with 0,0 as its top left-hand corner with the items ‘Continue’ and ‘Exit’, with the shortcut keys Ctrl+C and Ctrl+E respectively.

See also mCARD.

❸ NEWOBJ

Usage: `pobj%=NEWOBJ(num%,clnum%)`

Create a new object by category number `num%` belonging to the class `clnum%`, returning the object handle on success or zero if out of memory.

❺ The Series 5 supports use of OPXs only. See the ‘OPX.pdf’ document for further details.

❸ NEWOBJH

Usage: `pobj%=NEWOBJH(cat%,clnum%)`

Create a new object by category handle `cat%` belonging to the class `clnum%`, returning the object handle on success or zero if out of memory.

❺ The Series 5 supports use of OPXs only. See the ‘OPX.pdf’ document for further details.

NEXT

Usage: `NEXT`

Positions to the next record in the current data file.

If NEXT is used after the end of a file has been reached, no error is reported but the current record is null and the EOF function returns true.

NUM$

Usage: `n$=NUM$(x,y%)`

Returns a string representation of the integer part of the floating-point number `x`, rounded to the nearest whole number. The string will be up to `y%` characters wide.

- If `y%` is negative then the string is right-justified for example `NUM$(1.9,-3)` returns “ 2” where there are two spaces to the left of the `2`.
- If `y%` is positive no spaces are added: e.g. `NUM$(-3.7,3)` returns “`-4`”.
- If the string returned to `n$` will not fit in the width `y%`, then the string will just be asterisks; for example `NUM$(256.99,2)` returns “`**`”.

See also FIX$, GEN$, SCI$.

③ ODBINFO

Usage: `ODBINFO var info%()`

Provided for advanced use only, this keyword allows you to use OS and CALL to access data file interrupt functions not accessible with OPL keywords.

See the ‘Advanced.pdf’ document for more details.

❺ The Series 5 supports calls to the operating system using OPXs. Full description of their design is beyond the scope of this manual and is documented instead in the EPOC32 C++ Software Development Kit (SDK) which is available from Psion Software plc. See the ‘OPX.pdf’ document for details of built-in OPXs.

OFF

Usage: `OFF`

or `OFF x%`

Switches the Psion off.

When you switch back on, the statement following the OFF command is executed, for example:

```
OFF :PRINT “Hello again”
```

If you specify an integer, `x%`, greater than 2 (between 2 and 16383 on the Series 3c; 16383 is about 4.5 hours), the machine switches off for that number of seconds and then automatically turns back on and continues with the next line of the program. However, during this time the machine may be switched on by an alarm, and of course you can turn it on as usual.

❺ The minimum time to switch off is 5 seconds on the Series 5. EPOC32 also prevents switch off if there’s an absolute timer outstanding and due to go off in less than 5 seconds.

⚠ Be careful how you use this command. If, due to a programming mistake, a program uses OFF in a loop, you may find it impossible to switch the Psion back on, and may have to reset the computer.

ONERR

Usage: `ONERR label` *or* `ONERR label::`

```
...
ONERR OFF
```

`ONERR label::` establishes an error handler in a procedure. When an error is raised, the program jumps to the `label::` instead of the program stopping and an error message being displayed.

❺ The label may be up to 32 characters long starting with a letter or an underscore.

❸ The label may be up to 8 characters long starting with a letter.

It ends with a double colon (`::`), although you don’t need to use this in the ONERR statement.

ONERR OFF disables the ONERR command, so that any errors occurring after the ONERR OFF statement no longer jump to the label.

It is advisable to use the command ONERR OFF immediately after the `label::` which starts the error handling code.

See the ‘Errors.pdf’ document for full details.

OPEN

❺ Usage: `OPEN query$,log,f1,f2,...`

Opens an existing table (or a ‘view’ of a table) from an existing database, giving it the logical view name `log` and handles for the fields `f1`, `f2`. `log` can be any letter in the range `A` to `Z`.

`query$` specifies the database file, the required table and fields to be selected.

For example:

```
OPEN “clients SELECT name, tel FROM phone”,D,n$,t$
```

The database name here is `clients` and the table name is `phone`. The field names are enclosed by the keywords SELECT and FROM and their types should correspond with the list of handles (i.e. `n$` indicates that the `name` field is a string).

Replacing the list of field names with `*` selects all the fields from the table.

`query$` is also used to specify an ordered view and if a suitable index has been created, then it will be used. See ‘Database OPX’ in the ‘OPX.pdf’ document. For example,

```
OPEN “people SELECT name,number FROM phoneBook ORDER BY name ASC,
     number DESC”,G,n$,num%
```

would open a view with `name` fields in ascending alphabetical order and if any names were the same then the number field would be used to order these records in descending numerical order.

If the specification of the database includes embedded spaces, for example in the name of the folder, the name must be enclosed in quotes, so for example the following correctly fails:

```
OPEN “c:\folder with spaces\file with spaces”,a,name$
```

whereas the following works:

```
OPEN “””c:\folder with spaces\file with spaces”””,a,name$
```

COMPATIBILITY WITH THE SERIES 3C

As on the Series 3c (see below), `query$` may contain just the filename. In this case a table with the default name `Table1` would be opened if it exists. The field names would then be unimportant as access will be given to as many fields as there are supplied handles. The type indicators on the field handles must match the types of the fields.

❸ Usage: `OPEN file$,log,f1,f2...`

Opens an existing data file `file$`, giving it the logical file name `log`, and giving the fields the names `f1`, `f2`…

You need only specify those fields which you intend to update or append, though you cannot miss out a field.

The opened file is then referred to within the program by its logical name (`A`, `B`, `C` or `D`).

Up to 4 files can be open at once.

Example:

```
OPEN “clients”,A,name$,addr$
```

See also the ‘Data File Handling’ and ‘Series 5 Database Handling’ sections of the ‘Database.pdf’ document.

See also CREATE, USE and OPENR.

OPENR

Usage: ❺ `OPEN query$,log,f1,f2,...`

❸ `OPENR file$,log,f1,f2...`

This command works exactly like OPEN except that the opened file is read-only - in other words, you cannot APPEND, UPDATE or PUT the records it contains.

This means that you can run two separate programs at the same time, both sharing the same file.

❸ OS

Usage: `a%=OS(i%,addr1%)`

or `a%=OS(i%,addr1%,addr2%)`

Calls the Operating System interrupt `i%`, reading the values of all returned 8086 registers and flags. The CALL function, although simpler to use, does not allow the AL register to be passed and no flags are returned, making it suitable only for certain interrupts.

The input registers are passed at the address `addr1%`. The output registers are returned at the address `addr2%` if supplied, otherwise they are returned at `addr1%`. Both addresses can be of an array, or of six consecutive integers.

Register values are stored sequentially as 6 integers and represent the register values **in this order**: AX, BX, CX, DX, SI and DI. The interrupt's function number, if required, is passed in AH.

The output array must be large enough to store the 6 integers returned in all cases, irrespective of the interrupt being called.

The value returned by OS is the flags register. The Carry flag, which is relevant in most cases, is in bit 0 of the returned value, so `(a% AND 1)` will be 'True' if Carry is set. Similarly, the Zero flag is in bit 6, the Sign flag in bit 7 and the Overflow flag in bit 10.

For example, to find `COS(pi/4)`:

```
PROC add:
  LOCAL a%,b%,c%,d%,si%,di%
  LOCAL result,cosArg,flags%
  cosArg=pi/4
  si%=ADDR(cosArg)
  di%=ADDR(result)
  ax%=$0100                    REM AH=1 for cosine
  flags%=OS(140,addr(ax%))
ENDP
```

The OS function requires **extensive** knowledge of the Operating System and related programming techniques.

See also CALL.

❺ The Series 5 supports calls to the operating system using OPXs. Full description of their design is beyond the scope of this manual and is documented instead in the EPOC32 C++ Software Development Kit (SDK) which is available from Psion Software plc. See the 'OPX.pdf' document for details of built-in OPXs.

PARSE$

Usage: `p$=PARSE$(f$,rel$,var off%())`

Returns a full file specification from the filename `f$`, filling in any missing information from `rel$`.

The offsets to the filename components in the returned string is returned in `off%()` which must be declared with at least 6 integers:

off%(1)	filing system offset (1 always)
off%(2)	device offset (1 always on Series 5 since filing system is not a component of filenames on the Series 5)
off%(3)	path offset
off%(4)	filename offset
off%(5)	file extension offset
off%(6)	flags for wildcards in returned string

The flag values in `offset%(6)` are:

0	no wildcards
1	wildcard in filename
2	wildcard in file extension
3	wildcard in both

❺ Constants for the array subscripts and the 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.

If `rel$` is not itself a complete file specification, the current filing system, device and/or path are used as necessary to fill in the missing parts.

`f$` and `rel$` should be separate strings.

❺ Example:

```
p$=PARSE$(“NEW”,“C:\Documents\*.MBM”,x%())
```

sets `p$` to C`:\Documents\NEW.MBM` and `x%()` to `(1,1,3,14,17,0)`.

❸ Example:

```
p$=PARSE$(“NEW”,“LOC::M:\ODB\*.ODB”,x%())
```

sets `p$` to `LOC::M:\ODB\NEW.ODB` and `x%()` to `(1,6,8,13,16,0)`.

❸ PATH

Usage: `PATH name$`

Gives the folder (directory on the Series 3c) to use for an OPA's files.

This can only be used between APP and ENDA.

See the 'Advanced.pdf' document for more details of OPAs.

❺ OPL Applications do not have paths on the Series 5, so this command is not used.

PAUSE

Usage: `PAUSE x%`

Pauses the program for a certain time, depending on the value of `x%`:

`x%`	*result*
0	waits for a key to be pressed.
+ve	pauses for `x%` twentieths of a second.
-ve	pauses for `x%` twentieths of a second or until a key is pressed.

So `PAUSE 100` would make the program pause for 100/20 = 5 seconds, and `PAUSE -100` would make the program pause for 5 seconds or until a key is pressed.

If `x%` is less than or equal to 0, a GET, GET$, KEY or KEY$ will return the key press which terminated the pause. If you are not interested in this keypress, but in the one which follows it, clear the buffer after the PAUSE with a single KEY function: `PAUSE -10 :KEY`

You should be especially careful about this if `x%` is negative, since then you cannot tell whether the pause was terminated by a keypress or by the time running out.

PAUSE should not be used in conjunction with GETEVENT or GETEVENT32 because events are discarded by PAUSE.

PEEK FUNCTIONS

The PEEK functions find the values stored in specific bytes.

❺ `p%=PEEKB(x&)`

❸ `p%=PEEKB(x%)`

Returns the integer value of the byte at address `x&` (`x%`)

❺ `p%=PEEKW(x&)`

❸ `p%=PEEKW(x%)`

Returns the integer at address `x&` (`x%`)

❺ `p&=PEEKL(x&)`

❸ `p&=PEEKL(x%)`

Returns the long integer value at address `x&` (`x%`)

❺ `p=PEEKF(x&)`

❸ `p=PEEKF(x%)`

Returns the floating-point value at address `x&` (`x%`)

❺ `p$=PEEK$(x&)`

❸ `p$=PEEK$(x%)`

Returns the string at address `x&` (`x%`)

Usually you would find out the byte address with the ADDR function. For example, if `var%` has the value 7, `PEEKW(ADDR(var%))` returns `7`.

The different types are stored in different ways across bytes:

- *Integers* are stored in two bytes. The first byte is the least significant byte, for example:

  1 0 = 1

  0 1 = 256

  ADDR returns the address of the first (least significant) byte.

- *Long integers* are stored in four bytes, the least significant first and the most significant last, for example:

  0 0 1 0 = 65536

  ADDR returns the address of the first (least significant) byte.

- *Strings* are stored with one character per byte, with a leading byte containing the string length, e.g.:

  3 65 66 67 = “ABC”

  Each letter is stored as its character code - for example, A as 65.

  For example, if var$=“ABC”, PEEK$(ADDR(var$)) will return the string ABC. ADDR returns the address of the length byte.

- *Floating-point numbers* are stored under IEEE format, across eight bytes. PEEKF automatically reads all eight bytes and returns the number as a floating-point. For example if var=1.3 then PEEKF(ADDR(var)) returns 1.3.

You can use ADDR to find the address of the first element in an array, for example `ADDR(x%()`, you can also specify individual elements of the array, for example `ADDR(x%(2))`.

See also the POKE commands and ADDR. See also ‘Series 5 32-bit addressing’ in the ‘Advanced.pdf’ document.

PI

Usage: `p=PI`

Returns the value of π (3.14...).

❺ POINTERFILTER

Usage: `POINTERFILTER filter%,mask%`

Filters pointer events in the current window out or back in. Add the following flags together to achieve the desired `filter%` and `mask%`:

event	*value*
none	0
enter/exit	1
drag	4

Constants for these 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.

The bits set in `filter%` specify the settings to be used, 1 to filter out the event and 0 to remove the filter. Only those bits set in `mask%` will be used for filtering. This allows the current setting of a particular bit to be left unchanged if that bit is zero in the mask. (i.e. `mask%` dictates what to change and `filter%` specifies the setting to which it should be changed).

For example:

```
mask%=5
```

```
REM  =1+4 - allows enter/exit and drag settings to be changed
POINTERFILTER 1,mask% REM filters out enter/exit, but not dragging
...
POINTERFILTER 4,mask% REM filters out drag and reinstates enter/exit
```

Initially the events are not filtered out.

See also GETEVENT32, GETEVENTA32.

POKE COMMANDS

The POKE commands store values in specific bytes.

❺ `POKEB x&,y%`

❸ `POKEB x%,y%`

Stores the integer value `y%` (less than 256) in the single byte at address `x&` (`x%`)

❺ `POKEW x&,y%`

❸ `POKEW x%,y%`

Stores the integer `y%` across two consecutive bytes, with the least significant byte in the lower address, that is `x&` (`x%`)

❺ `POKEL x&,y&`

❸ `POKEL x%,y&`

Stores the long-integer `y&` in bytes starting at address `x&` (`x%`)

❺ `POKEF x&,y`

❸ `POKEF x%,y`

Stores the floating-point value `y` in bytes starting at address `x&` (`x%`)

❺ `POKE$ x&,y$`

❸ `POKE$ x%,y$`

Stores the string `y$` in bytes starting at address `x&` (`x%`)

Use ADDR to find out the address of your declared variables.

⚠ Casual use of these commands can result in the loss of data in the Series 3c.

See PEEK for more details of how the different types are stored across bytes.

POS

Usage: `p%=POS`

❺ Returns the number of the current record in the current view. **You are advised to use bookmarks instead of POS on the Series 5.**

❸ Returns the number of the current record in the current data file, from 1 (the first record) upwards.

A Series 5 file has no limit on the number of records and Series 3c files can have up to 65534 records. However, integers can only be in the range -32768 to +32767. Record numbers above 32767 are therefore returned like this:

record value	*returned by POS*
32767	32767
32768	-32768
32769	-32767
32770	-32766
.	.
65534	-2

To display record numbers, you can use this check:

```
IF POS<0
     PRINT 65536+POS
     ELSE
     PRINT POS
ENDIF
```

❺ Note that on the Series 5 the number of the current record may be greater than or equal to 65535 and hence values may need to be truncated to fit into `p%`, giving inaccurate results. **You are particuarly advised to use bookmarks when dealing with a large number of records.** Note, however, that the value returned by POS can become inaccurate if used in conjunction with bookmarks and multiple views on a table. Accuracy can be restored by using FIRST or LAST on the current view.

See BOOKMARK, GOTOMARK, KILLMARK.

POSITION

Usage: `POSITION x%`

❺ Makes record number `x%` the current record in the current view. By using bookmarks and editing the same table via different views, positional accuracy can be lost and `POSITION x%` could access the wrong record. Accuracy can be restored by using FIRST or LAST on the current view.

POS and POSITION still exist mainly for reasons of compatibility with the Series 3c and **you are advised to use bookmarks instead on the Series 5.**

See BOOKMARK, GOTOMARK, KILLMARK.

Makes record number `x%` the current record in the current data file.

If x% is greater than the number of records in the file then the EOF function will return true.

❸ POSSPRITE

Usage: `POSSPRITE x%,y%`

Set the position of the current sprite to pixel `x%,y%`.

❺ On the Series 5, sprites are handled by a built-in OPX. See the ‘OPX.pdf’ document for more details.

PRINT

Usage: `PRINT` *`list of expressions`*

Displays a list of expressions on the screen. The list can be punctuated in one of these ways:

If items to be displayed are separated by commas, there is a space between them when displayed.

If they are separated by semicolons, there are no spaces.

Each PRINT statement starts a new line, unless the preceding PRINT ended with a semicolon or comma.

There can be as many items as you like in this list. A single PRINT on its own just moves to the next line.

Examples: On 1st January 1993,

code *display*

```
PRINT “TODAY is”,
PRINT DAY;”.”;MONTH;”.”;YEAR          TODAY is 1.1.1993

PRINT 1                               1
PRINT “Hello”                         Hello
PRINT “Number”,1                      Number 1
```

See also LPRINT, gUPDATE, gPRINT, gPRINTB, gPRINTCLIP, gXPRINT.

❺ PUT

Usage: `PUT`

Marks the end of a database’s INSERT or MODIFY phase and makes the changes permanent.

See INSERT, MODIFY, CANCEL.

RAD

Usage: `r=RAD(x)`

Converts `x` from degrees to radians.

All the trigonometric functions assume angles are specified in radians, but it may be easier for you to enter angles in degrees and then convert with RAD.

Example:

```
PROC xcosine:
  LOCAL angle
  PRINT “Angle (degrees)?:”;
  INPUT angle
  PRINT “COS of”,angle,”is”,
  angle=RAD(angle)
  PRINT COS(angle)
  GET
ENDP
```

(The formula used is `(PI*x)/180`).

To convert from radians to degrees use DEG.

RAISE

Usage: `RAISE x%`

Raises an error.

The error raised is error number `x%`. This may be one of the errors listed in the ‘Errors.pdf’ document, or a new error number defined by you.

The error is handled by the error processing mechanism currently in use — either OPL’s own, which stops the program and displays an error message, or the ONERR handler if you have ONERR on.

❺ TRAP RAISE

Usage: `TRAP RAISE x%`

Sets the value of ERR to `x%` and clears the trap flag.

For a full explanation of the use of RAISE, see the ‘Errors.pdf’ document.

RANDOMIZE

Usage: `RANDOMIZE x&`

Gives a ‘seed’ (start-value) for RND.

Successive calls of the RND function produce a sequence of random numbers. If you use RANDOMIZE to set the seed back to what it was at the beginning of the sequence, the same sequence will be repeated.

For example, you might want to use the same ‘random’ values to test new versions of a procedure. To do this, precede the RND statement with the statement `RANDOMIZE value`. Then to repeat the sequence, use `RANDOMIZE value` again.

Example:

```
PROC SEQ:
   LOCAL g$(1)
   WHILE 1
     PRINT “S: set seed to 1”
     PRINT “Q: quit”
     PRINT “other key: continue”
     g$=UPPER$(GET$)
     IF g$=“Q”
          BREAK
     ELSEIF g$=“S”
          PRINT “Setting seed to 1”
          RANDOMIZE 1
          PRINT “First random no:”
     ELSE
          PRINT “Next random no:”
     ENDIF
     PRINT RND
   ENDWH
ENDP
```

REALLOC

Usage: ❺ `pcelln&=REALLOC(pcell&,size&)`

❸ `pcelln%=REALLOC(pcell%,size%)`

Change the size of a previously allocated cell at `pcell&` (`pcell%`) to `size&` (`size%`), returning the new cell address or zero if there is not enough memory.

❺ Cells are allocated lengths that are the smallest multiple of four greater than the size requested. An error will be raised if the cell address argument is not in the range known by the heap.

See also SETFLAGS if you require the 64K limit to be enforced on the Series 5. If the flag is set to restrict the limit, `pcelln&` is guaranteed to fit into an integer.

See also the ‘Advanced.pdf’ document.

❸ RECSIZE

Usage: `r%=RECSIZE`

Returns the number of bytes occupied by the current record.

Use this function to check that a record may have data added to it without overstepping the 1022-character limit.

Example:

```
PROC rectest:
  LOCAL n$(20)
  OPEN “name”,A,name$
  PRINT “Enter name:”,
  INPUT n$
  IF RECSIZE<=(1022-LEN(n$))
    A.name$=n$
    APPEND
  ELSE
    PRINT “Won’t fit in record”
  ENDIF
ENDP
```

❺ This function is not required on the Series 5 because the character limit is larger than the OPL record length (i.e. 32×256 characters, the number of fields multiplied by the maximum string length).

REM

Usage: `REM` *`text`*

Precedes a remark you include to explain how a program works. All text after the REM up to the end of the line is ignored.

When you use REM at the end of a line you need only precede it with a space, not a space and a colon.

Examples:

```
INPUT a :b=a*.175  REM b=TAX
```

```
INPUT a :b=a*.175 :REM b=TAX
```

RENAME

Usage: `RENAME file1$,file2$`

Renames `file1$` as `file2$`. You can rename any type of file.

You cannot use wildcards.

You can rename across directories `RENAME “\dat\xyz.abc”,”\xyz.abc”` is OK. If you do this, you can choose whether or not to change the name of the file.

Example:

```
PRINT “Old name:” :INPUT a$
PRINT “New name:” :INPUT b$
RENAME a$,b$
```

REPT$

Usage: `r$=REPT$(a$,x%)`

Returns a string comprising `x%` repetitions of `a$`.

For example, if `a$="ex"`, `r$=REPT$(a$,5)` returns `exexexexex`.

RETURN

Usage: `RETURN`

or `RETURN` *`variable`*

Terminates the execution of a procedure and returns control to the point where that procedure was called (ENDP does this automatically).

`RETURN` *`variable`* does this as well, but also passes the value of *`variable`* back to the calling procedure. The variable may be of any type. You can return the value of any single array element - for example `RETURN x%(3)`. **You can only return one variable.**

RETURN on its own, and the default return through ENDP, causes the procedure to return the value 0 or a null string.

Example:

```
PROC price:
  LOCAL x
  PRINT "Enter price:",
  INPUT x
  x=tax:(x)
  PRINT x
  GET
ENDP

PROC tax:(price)
  RETURN price+17.5
ENDP
```

RIGHT$

Usage: `r$=RIGHT$(a$,x%)`

Returns the rightmost `x%` characters of `a$`.

Example:

```
PRINT "Enter name/ref",
INPUT c$
ref$=RIGHT$(c$,4)
name$=LEFT$(c$,LEN(c$)-4)
```

❺ ROLLBACK

Usage: `ROLLBACK`

Cancels the current transaction on the current view. Changes made to the database with respect to this particular view since BEGINTRANS was called will be discarded.

See also BEGINTRANS, COMMITTRANS.

RMDIR

Usage: `RMDIR str$`

Removes the directory given by `str$`. You can only remove empty directories.

RND

Usage: `r=RND`

Returns a random floating-point number in the range 0 (inclusive) to 1 (exclusive).

To produce random numbers between `1` and `n` e.g. between 1 and 6 for a dice use the following statement: `f%=1+INT(RND*n)`

RND produces a different number every time it is called within a program. A fixed sequence can be generated by using RANDOMIZE. You might begin by using RANDOMIZE with an argument generated from MINUTE and SECOND (or similar), to seed the sequence differently each time.

Example:

```
PROC rndvals:
  LOCAL i%
  PRINT “Random test values:”
  DO
    PRINT RND
    i%=i%+1
    GET
  UNTIL i%=10
ENDP
```

SCI$

Usage: `s$=SCI$(x,y%,z%)`

Returns a string representation of `x` in scientific format, to `y%` decimal places and up to `z%` characters wide.

Examples:

`SCI$(123456,2,8)=“1.23E+05”`

`SCI$(1,2,8)=“1.00E+00”`

`SCI$(1234567,1,-8)=“ 1.2E+06”`

If the number does not fit in the width specified then the returned string contains asterisks.

If `z%` is negative then the string is right-justified.

See also FIX$, GEN$, NUM$.

SCREEN

Usage: `SCREEN width%,height%`

or `SCREEN width%,height%,x%,y%`

Changes the size of the window in which text is displayed. `x%,y%` specify the character position of the top left corner; if they are not given, the text window is centred in the screen.

An OPL program can initially display text to the whole screen.

See SCREENINFO.

SCREENINFO

Usage: `SCREENINFO var info%()`

Gets information on the text screen (as used by PRINT, SCREEN etc.)

This keyword allows you to mix text and graphics. It is required because while the default window is the same size as the physical screen, the text screen is slightly smaller and is centred in the default window. The few pixels gaps around the text screen, referred to as the left and top margins, depend on the font in use.

On return, the array `info%()`, which must have at least 10 elements, contains this information:

`info%(1)` left margin in pixels

`info%(2)` top margin in pixels

`info%(3)` text screen width in character units

`info%(4)` text screen height in character units

`info%(5)` reserved (window server id for default window)

❺ The font ID is a 32-bit integer on the Series 5 and therefore would not fit into a single element of `info%()`. Hence, the least significant 16 bits of the font ID are returned to `info%(9)` and the most significant 16 bits to `info%(10)`.

`info%(6)` unused

`info%(7)` pixel width of text window character cell

`info%(8)` pixel height of text window line

`info%(9)` least significant 16 bits of the font ID

`info%(10)` most significant 16 bits of the font ID

Constants for these array subscripts 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.

❸ On the Series 3c, the font ID is returned to info%(6).

`info%(6)` font ID

`info%(7)` pixel width of text window character cell

`info%(8)` pixel height of text window line

`info%(9)`, `info%(10)` reserved

Initially SCREENINFO returns the values for the initial text screen. Subsequently any keyword which changes the size of the text screen font, such as FONT or SCREEN, will change some of these values and SCREENINFO should therefore be called again.

See also FONT, SCREEN.

SECOND

Usage: `s%=SECOND`

Returns the current time in seconds from the system clock (0 to 59).

E.g. at 6:00:33 `SECOND` returns `33`.

SECSTODATE

Usage:
```
SECSTODATE s&,var yr%,var mo%,var dy%,var hr%,var mn%,
                 var sc%,var yrday%
```

Sets the variables passed by reference to the date corresponding to `s&`, the number of seconds since 00:00 on 1/1/1970. `yrday%` is set to the day in the year (1-366).

`s&` is an **unsigned** long integer. To use values greater than +2147483647, subtract 4294967296 from the value.

See also DATETOSECS, HOUR, MINUTE, SECOND, dDATE, DAYS.

❸ SEND

Usage: `ret%=SEND(pobj%,m%,var p1,...)`

Send a message to the object `pobj%` to call the method number `m%`, passing between zero and three arguments (`p1`...) depending on the requirements of the method, and returning the value returned by the selected method.

❺ The Series 5 supports use of OPXs only. See the ‘OPX.pdf’ document for further details.

❺ SETDOC

Usage: `SETDOC file$`

Sets the file `file$` to be a document. This command should be called immediately before the creation of `file$` if it is to be recognised as a document. SETDOC may be used with the commands CREATE, gSAVEBIT and IOOPEN.

The string passed to SETDOC must be identical to the name passed to the following CREATE or gSAVEBIT otherwise a non-document file will be created. Example of document creation:

```
SETDOC “myfile”
CREATE “myfile”,a,a$,b$
```

SETDOC should also be called after successfully opening a document to allow the System screen to display the correct document name in its task list.

In case of failure in creating or opening the required file, you should take the following action:

- Creating - try to re-open the last file and if this fails display an appropriate error dialog and exit. On reopening, call SETDOC back to the original file so the Task list is correct.
- Opening - as for creating, but calling SETDOC again is not strictly required.

Database documents, created using CREATE, and multi-bitmap documents, created using gSAVEBIT, will automatically contain your application UID in the file header. For binary and text file documents created using IOOPEN and LOPEN, it is the programmer’s responsibility to save the appropriate header in the file. This is a fairly straight-forward process and the following suggests one way of finding out what the header should be:

1. Create a database or bitmap document in a test run of you application using SETDOC as shown above
2. Use a hex editor or hex dump program to find the 1st 16 bytes, or run the program below which reads the four long integer UIDs from the test document.
3. Write these four long integers to the start of the file you create using IOOPEN.

```
INCLUDE “Const.oph”
DECLARE EXTERNAL
EXTERNAL readUids:(file$)

PROC main:
  LOCAL f$(255)
  WHILE 1
    dINIT “Show UIDs in document header”
    dPOSITION 1,0
    dFILE f$,”Document,Folder,Drive”,0
    IF DIALOG=0
        RETURN
    ENDIF
    readUids:(f$)
  ENDWH
ENDP

PROC readUids:(file$)
  LOCAL ret%,h%
  LOCAL uid&(4),i%

  ret%=IOOPEN(h%,file$, KIoOpenModeOpen% OR KIoOpenFormatBinary%)
  IF ret%>=0
    ret%=IOREAD(h%,ADDR(uid&()),16)
    PRINT “Reading “;file$
    IF ret%=16
        WHILE i%<4
            i%=i%+1
            print “  Uid”+num$(i%,1)+”=”,hex$(uid&(i%))
        ENDWH
    ELSE
        PRINT “  Error reading: “;
        IF ret%<0
            PRINT err$(ret%)
        ELSE
            PRINT “Read “;ret%;” bytes only (4 long integers required)”
        ENDIF
    ENDIF
    IOCLOSE(h%)
  ELSE
    PRINT “Error opening: “;ERR$(ret%)
  ENDIF
ENDP
```

Creating text file documents using IOOPEN or LOPEN has two special requirements:

- You will need to save the required header as the first text record. This will insert the standard text file line delimiters CR LF (hex 0D 0A) at the end of the record.

- The specific 16 bytes required for your application may itself however contain CR LF. Since you should know when this is the case, you will need to read records until you have reached byte 16 in the document. This is clearly not a desirable state of affairs but is inescapable given that text files were not designed to have headers. It is recommended that you request a new UID for your application if it contains CR LF.

See the ‘Advanced.pdf’ document.

See also GETDOC$.

❺ SETFLAGS

Usage: `SETFLAGS flags&`

Sets flags to produce various effects when running programs. Use CLEARFLAGS to clear any flags which have been set. The effects which can be achieved are as follows:

`flags&`	*effect*
1	restricts the memory available to your application to 64K, emulating the Series 3c. This setting should be used at the beginning of your program only, if required. Changing this setting repeatedly will have unpredictable effects.
2	enables auto-compaction on closing databases. This can be slow, but it is advisable to use this setting when lots of changes have been made to a database.
4	enables raising of overflow errors when floating-point values are greater than or equal to 1.0E+100 in magnitude, instead of allowing 3-digit exponents (for backwards compatibility).
$10000	enables GETEVENT, GETEVENT32 and GETEVENT32A to return the event code $403 to ev&(1) when the machine switches on

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.

By default these flags are cleared.

See the ‘Advanced.pdf’ document, and the ‘Series 5 Database Handling’ section of the ‘Database.pdf’ document.

See also GETEVENT32, CLEARFLAGS.

❸ SETNAME

Usage: `SETNAME name$`

Sets the name of the running OPA to `name$` and redraws any status window, using that name below the icon.

❺ SETDOC serves a similar purpose on the Series 5.

SETPATH

Usage: `SETPATH name$`

Sets the current path for file access for example,

❺ `SETPATH “C:\Documents\”`.

❸ `SETPATH “a:\docs\”`.

LOADM continues to use the path of the initial program, but all other file access will use the new path.

SIN

Usage: `s=SIN(angle)`

Returns the sine of `angle`, an angle expressed in radians.

To convert from degrees to radians, use the RAD function.

SPACE

Usage: `s&=SPACE`

Returns the number of free bytes on the device on which the current (open) data file is held.

Example:

❺

```
PROC stock:
  OPEN “c:\stock\stock”,A,a$,b%
  PRINT SPACE,“bytes free on C:”
ENDP
```

❸

```
PROC stock:
  OPEN “a:\stock”,A,a$,b%
  WHILE 1
       PRINT “Item name:”;
       INPUT A.a$
       PRINT “Number:”;
       INPUT A.b%
       IF RECSIZE>SPACE
            PRINT “Disk full”
            CLOSE
            BREAK
```

```
      ELSE
           APPEND
      ENDIF
 ENDWH
ENDP
```

SQR

Usage: `s=SQR(x)`

Returns the square root of `x`.

❸ STATUSWIN

Usage: any of

```
STATUSWIN ON,type%
STATUSWIN ON
STATUSWIN OFF
```

Displays or removes a ‘permanent’ status window.

If `type%=1` the small status window is shown. If `type%=2` the large status window is shown. STATUSWIN ON on its own displays an appropriate status window; on the Series 3c this will always be the large status window.

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

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

❺ The Series 5 has no status windows, but they are replaced by the toolbar. See the ‘Friendlier Interaction’ section of the ‘GUI.pdf’ document for more details.

❸ STATWININFO

Usage: `t%=STATWININFO(type%,var xy%())`

Sets `xy%(1)`,`xy%(2)`, `xy%(3)` and `xy%(4)` to the top left x, top left y, width and height respectively of the specified type of status window. `type%=1` is the small status window; `type%=2` is the large status window; `type%=3` is the Series 3 compatibility mode status window; `type%=-1` is whichever status window is **current**.

STATWININFO returns `t%`, the type of the **current** status window (with values as for `type%`, or zero if there is no current status window).

❺ The Series 5 has no status windows, but they are replaced by the toolbar. See the ‘Friendlier Interaction’ section of the ‘GUI.pdf’ document for more details.

STD

Usage: `s=STD(list)`

or `s=STD(array(),element)`

Returns the standard deviation of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on for example `m=STD(arr(),3)` would return the standard deviation of elements `arr(1)`, `arr(2)` and `arr(3)`.

This function gives the sample standard deviation, using the formula:

$$\sqrt{\left(\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2 / (n-1)\right)}$$

where $\bar{x}$ means $\sum_{i=1}^{n} x_i / n$. To convert to population standard deviation, multiply the result by `SQR((n-1)/n)`.

STOP

Usage: `STOP`

Ends the running program.

❺ Note that STOP may not be used during an OPX callback and will raise the Series 5 error, ‘STOP used in callback’ if it is. See the ‘OPX.pdf’ document.

STYLE

Usage: `STYLE style%`

Sets the text window character style. `style%` can be 2 for underlined, or 4 for inverse.

See ‘The text and graphics windows’ section at the end of the ‘Graphics’ part of the ‘GUI.pdf’ document for more details.

SUM

Usage: `s=SUM(list)`

or `s=SUM(array(),element)`

Returns the sum of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on for example `m=SUM(arr(),3)` would return the sum of elements `arr(1)`, `arr(2)` and `arr(3)`.

TAN

Usage: `t=TAN(angle)`

Returns the tangent of `angle`, an angle expressed in radians.

To convert from radians to degrees, use the DEG function.

TESTEVENT

Usage: `t%=TESTEVENT`

Returns ‘True’ if an event has occurred, otherwise returns ‘False’. The event is not read by TESTEVENT - it may be read with GETEVENT, or GETEVENT32 or GETEVENTA32 on the Series 5.

⚠ On the Series 5, it is recommended that you use either GETEVENT32 or GETEVENTA32 **without TESTEVENT** as TESTEVENT may use a lot of power, especially when used in a loop as will often be the case.

TRAP

Usage: `TRAP` *`command`*

TRAP is an error handling command. It may precede any of these commands:

DATA FILE COMMANDS

APPEND, UPDATE, BACK, NEXT, LAST, FIRST, POSITION, USE, CREATE, OPEN, OPENR, CLOSE, DELETE

❺ MODIFY, INSERT, PUT, CANCEL

FILE COMMANDS

COPY, ERASE, RENAME, LOPEN, LCLOSE, LOADM, UNLOADM

❸ COMPRESS

DIRECTORY COMMANDS

MKDIR, RMDIR

DATA ENTRY COMMANDS

EDIT, INPUT

GRAPHICS COMMANDS

gSAVEBIT, gCLOSE, gUSE, gUNLOADFONT, gFONT, gPATT, gCOPY

For example, `TRAP FIRST`.

Any error resulting from the execution of the command will be trapped. Program execution will continue at the statement after the TRAP statement, but ERR will be set to the error code.

TRAP overrides any ONERR.

⑤ TRAP RAISE

Usage: `TRAP RAISE x%`

Sets the value of ERR to `x%` and clears the trap flag.

See the ‘Errors.pdf’ document for further details of TRAP usage.

③ TYPE

Usage: `TYPE num%`

Sets the type of an OPA, from 0 to 4, with `num%`. On the Series 3c you should set `num%` from $1000 to $1004 to set type 0 to 4 respectively; the $1000 allows a 48x48 black/grey icon to be used.

This can only be used between APP and ENDA.

See the ‘Advanced.pdf’ document for more details of OPAs.

❺ OPL Applications do not have types on the Series 5, but FLAGS provides similar functionality.

UADD

Usage: `i%=UADD(val1%, val2%)`

Add `val1%` and `val2%`, as if both were unsigned integers with values from 0 to 65535. Prevents integer overflow for pointer arithmetic on the Series 3c e.g. `UADD(ADDR(text$),1)` should be used instead of `ADDR(text$)+1`.

One argument would normally be a pointer and the other an offset expression.

❺ Note that UADD and USUB should **not** be used on the Series 5 for pointer arithmetic unless SETFLAGS has been used to enforce the 64K memory limit. In general, long intger arithmetic should be used for pointer arithmetic on the Series 5.

See also USUB.

❸ UNLOADLIB

Usage: `ret%=UNLOADLIB(var cat%)`

Unload a DYL from memory. If successful, returns zero.

❺ The Series 5 supports use of OPXs only. See the ‘OPX.pdf’ document for further details.

UNLOADM

Usage: `UNLOADM module$`

Removes from memory the module `module$` loaded with LOADM.

`module$` is a string containing the name of the translated module.

The procedures in an unloaded module cannot then be called by another procedure.

UNLOADM causes any procedures in the module that are not still running to be unloaded from memory too. Running procedures are unloaded on return. It is considered bad practice, however, to use UNLOADM on a module with procedures that are still running. On the Series 3c, the module name and procedure name will not be available for any untrapped error message.

❺ Once LOADM has been called, procedures loaded stay in memory until the module is unloaded, so significantly more memory can be used than on the Series 3c, where procedures are unloaded when the cache is full (or on return if caching is not used).

UNTIL

See DO.

UPDATE

Usage: `UPDATE`

Deletes the current record in the current data file and saves the current field values as a new record at the end of the file.

This record, now the last in the file, remains the current record.

Example:

```
A.count=129
A.name$=”Brown”
UPDATE
```

Use APPEND to save the current field values as a new record.

❺ **MODIFY, PUT and CANCEL should normally be used instead of UPDATE on the Series 5.**

UPPER$

Usage: `u$=UPPER$(a$)`

Converts any lower case characters in `a$` to upper case, and returns the completely upper case string.

Example:

```
...
CLS :PRINT “Y to continue”
PRINT “or N to stop.”
g$=UPPER$(GET$)
IF g$=“Y”
     nextproc:
ELSEIF g$=“N”
     RETURN
ENDIF
...
```

Use LOWER$ to convert to lower case.

USE

Usage: `USE` *`logical name`*

Selects the data file with the given *logical name* (`A-Z` for the Series 5, `A`, `B`, `C` or `D` for the Series 3c). The file must previously have been opened with OPEN, OPENR or CREATE and not yet be closed.

All the record handling commands (such as POSITION and UPDATE, and GOTOMARK, INSERT, MODIFY, CANCEL and PUT on the Series 5) then operate on this file.

❸ USR

Usage: `u%=USR(pc%,ax%,bx%,cx%,dx%)`

Executes your machine code, returning an integer. The USR code (i.e. the assembler code you have written) **must return with a far RET**, otherwise the program will crash.

The values of `ax%,bx%`... are passed to the AX, BX... 8086 registers. The microprocessor then executes the machine code starting at `pc%`. At the end of the routine, the value in the AX register is passed back to `u%`.

⚠ Casual use of this function can result in the loss of data in the Psion.

This example shows a simple operation, ending with a far RET:

```
PROC trivial:
  LOCAL t%(2),u%,ax%
  t%(1)=$c032                  REM xor al,al
  t%(2)=$cb                    REM retf
  ax%=$1ab
  u%=usr(addr(t%(1)),ax%,0,0,0)  REM returns (ax% AND $FF00)
  PRINT u%                     REM 256 ($100)
  GET
ENDP
```

See also USR$, ADDR, PEEK, POKE.

❸ USR$

Usage: `u$=USR$(pc%,ax%,bx%,cx%,dx%)`

Executes your machine code, returning a string. The USR$ code you have written **must return with a far RET**, otherwise the program will crash.

The values of `ax%,bx%`... are passed to the ax, bx... 8086 registers. The microprocessor then executes the machine code starting at `pc%`. At the end of the routine, the value in the ax register must point to a length-byte preceded string. This string is then copied to `u$`.

⚠ Casual use of this function can result in the loss of data in the Psion.

See USR for an example. See also ADDR, PEEK, POKE.

USUB

Usage: `i%=USUB(val1%,val2%)`

Subtract `val2%` from `val1%`, as if both were unsigned integers with values from 0 to 65535. Prevents integer overflow for pointer arithmetic on the Series 3c.

❺ Note that UADD and USUB should **not** be used for pointer arithmetic on the Series 5 unless SETFLAGS has been used to enforce the 64K memory limit. In general long integer arithmetic should be used.

See also UADD.

VAL

Usage: `v=VAL(`*`numeric string`*`)`

Returns the floating-point number corresponding to a numeric string.

The string must be a valid number e.g. not `“5.6.7”` or `“196f”`. Expressions, such as `“45.6*3.1”`, are not allowed. Scientific notation such as `“1.3E10”`, is OK.

E.g. `VAL(“470.0”)` returns `470`

See also EVAL.

VAR

Usage: `v=VAR(list)`

or `v=VAR(array(),element)`

Returns the variance of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on for example `m=VAR(arr(),3)` would return the variance of elements `arr(1)`, `arr(2)` and `arr(3)`.

This function gives the sample variance, using the formula:

$\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2 / (n-1)$ where $\bar{x}$ means $\sum_{i=1}^{n} x_i / n$. To convert to population variance, multiply the result by `(n-1)/n`

VECTOR

Usage:
```
VECTOR i%
label1,label2,...,labelN
ENDV
```

`VECTOR i%` jumps to label number `i%` in the list. If `i%` is 1 this will be the first label, and so on. The list is terminated by the ENDV statement. The list may spread over several lines, with a comma separating labels in any one line but no comma at the end of each line.

If `i%` is not in the range 1 to *N*, where *N* is the number of labels, the program continues with the statement after the `ENDV` statement.

See also GOTO.

WEEK

Usage: `w%=WEEK(day%,month%,year%)`

Returns the week number in which the specified day falls, as an integer between 1 and 53.

`day%` must be between 1 and 31, `month%` between 1 and 12, `year%` between 0 and 9999 (1900 and 2155 on the Series 3c).

Each week is taken to begin on the ‘Start of week’ day, as specified in the Control Panel on the Series 5 or the Time application on the Series 3c. When a year begins on a different day to the start of the week, it counts as week 1 if there are four or more days before the next week starts.

❺ The System setting of the ‘Start of week’ may be checked from inside OPL by using the LCSTARTOFWEEK&: procedure in the Date OPX. The week number in the year may also be calculated by different rules and also with your own choice of the start of year by using the procedure DTWEEKNOINYEAR&: in Date OPX. See the ‘OPX.pdf’ document for more details.

WHILE...ENDWH

Usage:
```
WHILE expression
  ...
ENDWH
```

Repeatedly performs the set of instructions between the WHILE and the ENDWH statement, so long as *`expression`* returns logical true non-zero.

If *`expression`* is not true, the program jumps to the line after the ENDWH statement.

Every WHILE must be closed with a matching ENDWH.

See also DO...UNTIL and the ‘Loops and Branches’ section of the ‘Basics.pdf’ document.

YEAR

Usage: `y%=YEAR`

Returns the current year as an integer from the system clock.

For example, on 5th May 1997 YEAR returns `1997`.

INDEX

OPL

APPENDICES

CONTENTS

APPENDIX A

SUMMARY FOR EXPERIENCED OPL USERS

OPL has evolved from the Psion Organiser II through the MC and HC computers to the Series 3 and the Series 3a and then to the Series 3c and Siena and to the Series 5. This appendix gives a summary of the changes made from the Series 3a upwards. *For more details of the following topics and keywords, look them up in the ‘Alphabetical listing’ section of the ‘Glossary.pdf’ document.*

Bear in mind that some OPL keywords return or allow different values according to screen size and keyboard layout.

USING OPL ON THE SERIES 3A, SERIES 3C AND SIENA

OPL on the Series 3c is similar to that on the Series 3a except for some differences with serial cables.

Again the Siena is similar to these two, except for the size of its screen being around half that of the other two and also the fact that it does not have an SSD.

USING OPL ON THE SERIES 5

The principal design requirements of OPL for the Series 5 were:

- Compatibility with earlier versions of OPL.
- Provision of a mechanism for language extensions to replace direct calls to the operating system.
- Generally to take advantage of the abilities of EPOC32.

The major difference between this and other versions of OPL is that 32-bit rather than 16-bit addressing is used. This means that the arguments and return values of quite a number of keywords have changed from being integers to long integers.

Also graphics, menus, dialogs and database handling especially have been improved to take advantage of the abilities of EPOC32.

Below the removed, new and changed features are listed. For this appendix to provide detailed information would represent a repetition of much of what has been described in the main chapters of this manual and you should refer to these for full details (especially useful is the ‘Alphabetic Listing’ which includes full details of all keywords for both the Series 5 and the Series 3c).

The following keywords, available on earlier Psion machines (Series 3a, Series 3c and Siena), are no longer available on the Series 5:

- RECSIZE and COMPRESS. COMPACT replaces COMPRESS.
- gDRAWOBJECT.
- **gINFO. This is replaced by gINFO32.**
- STATUSWIN, STATWININFO, DIAMINIT and DIAMPOS. The Series 5 has no status windows. Toolbars are used instead See the ‘Toolbar Usage’ section in the ‘Friendlier Interaction’ chapter.
- TYPE, PATH and EXT. These were provided on earlier machines for use by the System screen only. The Series 5 does not require this information. On the Series 5, applications do not have types, application-specific paths or filename extensions. An application and its associated documents are identified by a UID (unique identifier) instead. (A new command FLAGS has a similar function to TYPE.) See the ‘Advanced Topics’ chapter.
- CMD$(4) and CMD$(5).
- SETNAME. The new SETDOC, serves a similar purpose, and should be called before saving your main document.
- Named Calculator memories and M0,...,M9. The Series 5 Calculator does not use OPL to evaluate expressions.
- CACHE, CACHETIDY, CACHEHDR and CACHEREC. On the Series 5 procedures are automatically cached.

- CREATESPRITE, APPENDSPRITE, CHANGESPRITE, DRAWSPRITE, POSSPRITE and CLOSESPRITE. Superior sprite-handling OPX functions are provided. See the ‘Sprite and Bitmap OPX’ section in the ‘Using OPXs on the Series 5’ chapter.
- OS and CALL. The Series 5 provides extensibility using special OPL DLLs. See the ‘Using OPXs on the Series 5’ chapter.
- USR and USR$.
- ODBINFO. This is implementation specific.
- LOADLIB, LINKLIB, UNLOADLIB, FINDLIB, GETLIBH, NEWOBJ, NEWOBJH, SEND, ENTERSEND and ENTERSEND0. OPXs are now used for calling language extensions and creating object instances. See the ‘Using OPXs on the Series 5’ chapter.

The following keywords have been added to OPL on the Series 5:

- DECLARE EXTERNAL, EXTERNAL, INCLUDE and CONST allow the use of header files which include the definition of constants and procedure prototypes.
- mCASC and mPOPUP provide new menu features.
- dCHECKBOX and dEDITMULTI provide new features for dialogs.
- DAYSTODATE allows easy conversion of “days since 1/1/1990” to a date.
- gCOLOR, gCIRCLE, gELLIPSE and gSETPENWIDTH provide new graphics functionality.
- gINFO32 replaces gINFO.
- SETFLAGS and CLEARFLAGS allow for Series 3a/Series 3c/Siena compatibility
- IOWAITSTAT32.

Database commands:

- DELETE allows deletion of a table.
- INSERT, MODIFY, PUT and CANCEL allow the building up and editing of databases.
- BOOKMARK, KILLMARK, GOTOMARK support the use of bookmarks in databases.
- BEGINTRANS, COMMITTRANS, INTRANS and ROLLBACK support transactions in databases.
- COMPACT replaces COMPRESS.

OPL applications:

- CAPTION and FLAGS allow definition of OPL applications; FLAGS is similar to TYPE.
- SETDOC and GETDOC$ allow files to be created as documents.
- GETEVENT32, GETEVENTA32 and POINTERFILTER provide increased support for handling of events, including pointer (pen) events.

OPL

Two other major additions have also been made to OPL. These are:

- Support for Toolbars (to replace status windows). See the ‘Toolbar Usage’ section in the ‘Friendlier Interaction’ chapter.
- Support for language extensions in separate EPOC32 DLLs called OPXs. See the ‘Using OPXs on the Series 5’ chapter.

The following keywords have undergone some changes, although many of these are compatible with earlier versions of OPL:

- ADDR, ALLOC, ADJUSTALLOC, REALLOC, LENALLOC and FREEALLOC
- APP and ICON
- CMD$(3)
- GETEVENT
- BUSY
- OFF
- SCREENINFO
- dBUTTONS, dCHOICE, dFILE, dINIT, dTEXT, dTIME and dXINPUT
- mCARD
- CLOSE, COUNT, CREATE, OPEN, POS and POSITION
- CURSOR, DEFAULTWIN, gBORDER, gBUTTON, gCLOCK, gCREATE, gCREATEBIT, gFONT, gGREY, gLINETO, gLINEBY, gLOADBIT, gSAVEBIT, gLOADFONT, gUNLOADFONT, gPEEKLINE and gXBORDER.

APPENDIX B

OPERATORS AND LOGICAL EXPRESSIONS

OPL

OPERATORS

These operators are available in OPL:

ARITHMETIC OPERATORS

`+`	add
`-`	subtract
`*`	multiply
`/`	divide
`**`	raise to a power
`-`	unary minus (in negative numbers for example, `-10`)
`%`	percent

COMPARISON OPERATORS

`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to
`=`	equal to
`<>`	not equal to

LOGICAL AND BITWISE OPERATORS

`AND`

`OR`

`NOT`

THE % OPERATOR

The percentage operator can be used in expressions like this:

Expression	*Meaning*	*Result*
`60+5%`	60 plus 5% of 60	63
`60-5%`	60 minus 5% of 60	57
`60*5%`	5% of 60	3
`60/5%`	of what number is 60 5%?	1200

It can also be used like this:

`105>5%`	what number, when increased by 5%, becomes 105?	100
`105<5%`	how much of 105 is a 5% increase?	5

Examples:

To add 15% tax to 345:

`345+15%`	Result = 396.75

To find out what the price was before tax:

`396.75>15%`	Result = 345

To find out how much of a total price is tax:

`396.75<15%`	Result = 51.75

❺ OTHER OPERATORS

On the Series 5, the System OPX provides two addtional operators:

`MOD&:(x&,y&)`	`x&` modulo `y&` (the remainder of x& divided by y&)
`XOR&:(x&,y&)`	exclusive OR of `x&` and `y&`

PRECEDENCE

Highest:	`**`
	`-` (unary minus)
	`NOT`
	`*   /`
	`+   -` (subtraction)
	`=   >   <   <>   >=   <=`
Lowest:	`AND  OR`

So `7+3*4` returns 19 (3 is first multiplied by 4, and 7 is added to the result) not 40 (4 times 10).

WHEN THERE IS EQUAL PRECEDENCE

In an expression where all operators have equal precedence, they are evaluated from left to right (with the exception of powers). For example, in `a+b-c`, `a` is added to `b` and then `c` is subtracted from the result.

Powers are evaluated from right to left for example, in `a%**b%**c%`, `b%` will first be raised to the power of `c%` and then `a%` will be raised to the power of the result.

CHANGING PRECEDENCE WITH BRACKETS

The result of an expression such as `a+b+c` is the same whether you first add `a` to `b`, or `b` to `c`. But how is `a+b*c/d` evaluated? You may want to use brackets to either:

- Make it obvious what the order of calculation is

or

- Change the order of calculation.

By default, `a+b*c/d` is evaluated in the order: `b` multiplied by `c`, then divided by `d`, then added to `a`. To perform the addition and the division before the multiplication, use brackets: `(a+b)*(c/d)`. When in doubt, simply use brackets.

PRECEDENCE OF INTEGER AND FLOATING-POINT VALUES

You are free to mix floating-point and integer values in expressions, but be aware how OPL handles the mix:

- In each part of the calculation, OPL uses the simplest arithmetic possible. Two integers will use integer arithmetic, and this can give unexpected results: **`7/2` gives the integer `3`**. Otherwise floating-point arithmetic is used (`7.0` is a floating-point number, so `7.0/2` gives the floating-point number `3.5`).

- Finally, the evaluated result of the right-hand side of an expression is automatically converted to the same type as the variable to which it is assigned.

For example, your procedure might include the expression `a%=b%+c` This is handled like this: `b%` is converted to floating-point and added to `c`. The resulting floating-point value is then automatically converted to an integer in order to be assigned to the integer variable `a%`.

Such conversions may produce odd results for example `a%=3.0*(7/2)` makes `a%=9`, but `a%=3.0*(7.0/2)` makes `a%=10`. OPL does not report this as an error, so it's up to you to ensure that it doesn't happen unless you want it to.

TYPE CONVERSIONS AND ROUNDING DOWN

There are three numeric types floating-point, integer and long integer. You can assign any of these types to any other. The value on the right-hand side will be automatically converted to the type of the variable on the left-hand side. For example:

- If you assign an integer value to a floating-point variable, there are no problems.

- If you assign a floating-point value to an integer variable, the value is converted to an integer, always rounded **towards zero** for example, if you declare `LOCAL c%` and then say `c%=3.75`, the value `3.75` is converted to the value `3`.

Rounding down towards zero can sometimes cause unusual results. For example, `a%=2.9` would give `a%` the value 2, and `a%=-2.3` would give `a%` the value `-2`.

When you run a module, if the left-hand side of an assignment has a narrower range than the right-hand side, you may get an error (for example, if you had `x%=a&` where `a&` had the value `320000`).

To control how floating-point numbers are rounded when converted, use the INT function.

LOGICAL EXPRESSIONS

The comparison operators and logical operators are based on the idea that a certain situation can be evaluated as either true or false. For example, if `a%=6` and `b%=8`, `a%>b%` would be 'False'.

These operators are useful for setting up alternative paths in your procedures. For example you could say:

```
IF salary<expenses
  doBad:
ELSE
  doGood:
ENDIF
```

You can also make use of the fact that the result of these logical expressions is represented by an integer:

- 'True' is represented by the integer -1

- 'False' is represented by the integer 0 (zero).

operator	*example*	*result returned*	*return value*
<	a<b	True if a less than b False if a greater than or equal to b	-1 0
>	a>b	True if a greater than b False if a less than or equal to b	-1 0
<=	a<=b	True if a less than or equal to b False if a greater than b	-1 0
=	a>=b	True if a greater than or equal to b False if a less than b	-1 0
<>	a<>b	True if a not equal to b False if a equal to b	-1 0
=	a=b	True if a equal to b False if a not equal to b	-1 0

These integers can be assigned to a variable or displayed on the screen to tell you whether a particular condition is true or false, or used in an IF statement.

For example, in a procedure you might arrive at two sub-totals, a and b. You want to find out which is the greater. So use the statement, PRINT a>b. If zero is displayed, a and b are equal or b is the larger number; if -1 is displayed, a>b is true a is the larger.

LOGICAL AND BITWISE OPERATORS

The operators AND, OR and NOT have different effects depending on whether they are used with floating-point numbers or integers:

WHEN USED WITH FLOATING-POINT NUMBERS

AND, OR and NOT are *logical operators*, and have the following effects:

example	*result*	*integer returned*
a AND b	True if both a and b are non-zero False if either a or b are zero	-1 0
a OR b	True if either a or b is non-zero False if both a and b are zero	-1 0
NOT a	True if a is zero False if a is non-zero	-1 0

WHEN USED WITH INTEGER OR LONG INTEGER VALUES

AND, OR and NOT are *bitwise operators*.

The way OPL holds integer numbers internally is as a binary code - 16-bit for integers, 32-bit for long integers. Bitwise means that an operation is performed on individual bits. A bit is *set* if it has the value 1, and *clear* if it has the value 0. Long integer values with AND, OR and NOT behave the same as integer values.

AND

Sets the result bit if both input bits are set, otherwise clears the result bit.

For example, the statement `PRINT 12 AND 10` displays 8. To understand this, write 12 and 10 in binary:

```
12   0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
10   0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
```

AND acts on each pair of bits. Thus, working from left to right discounting the first 12 bits (since `0 AND 0` gives 0):

```
1 AND 1   →   1
1 AND 0   →   0
0 AND 1   →   0
0 AND 0   →   0
```

The result is therefore the binary number 1000, or 8.

OR

Sets the result bit if either input bit is set, otherwise clears the result bit.

What result would the statement `PRINT 12 OR 10` give? Again, write down the numbers in binary and apply the operator to each pair of digits:

```
1 OR 1    →   1
1 OR 0    →   1
0 OR 1    →   1
0 OR 0    →   0
```

The result is the binary number 1110, or 14 in decimal.

NOT

*Sets the result bit if the input bit is **not** set, otherwise clears the result bit.*

NOT works on only one number. It returns the *one's complement*, i.e. it replaces 0s with 1s and 1s with 0s.

So since 7 is 0000000000000111, `NOT 7` is 1111111111111000. This is the binary representation of -8.

A quick way of calculating NOT for integers is to add 1 to the original number and reverse its sign. So `NOT 23` is -24, `NOT 0` is -1 and `NOT -1` is 0.

APPENDIX C

SERIAL/PARALLEL PORTS AND PRINTING

OPL

You can use LPRINT in an OPL program to send information (for printing or otherwise) to any of these devices:

- ❸ A parallel port, PAR:A
- A serial port, TTY:A
- A file on the Psion
- ❸ A file on an attached computer, e.g. on a PC or on an Apple Macintosh:

❸ OPL does not provide access to the advanced page formatting and font control features of the Series 3c.

❺ OPL provides more advanced formatting features and printing control using Printer OPX. See the ‘Using OPXs on the Series 5’ chapter for more details.

You can also read information from the serial port.

USING THE PARALLEL PORT ON THE SERIES 3C AND SIENA

In your OPL program, set up the port with the statement `LOPEN “PAR:A”`.

Provided the port is not already in use, the connection is now ready. LPRINT will send information down the parallel 3 Link lead for example, to an attached printer.

EXAMPLE

```
PROC prints:
   OPEN “clients”,A,a$
   LOPEN “PAR:A”
   PRINT “Printing...”
   DO
      IF LEN(A.a$)
            LPRINT A.a$
      ENDIF
      NEXT
   UNTIL EOF
   LPRINT CHR$(12); :LCLOSE
   PRINT “Finished” :GET
ENDP
```

USING THE SERIAL PORT

In your OPL program, set up the port with the statement `LOPEN “TTY:A”`.

Now LPRINT should send information down the serial link lead for example, to an attached printer. If it does not, the serial port settings are not correct.

SERIAL PORT SETTINGS

`LOPEN “TTY:A”` opens the serial port with the following default characteristics:

9600 baud

no parity

8 data bits

1 stop bit

RTS handshaking.

- If your printer (or other device) **does** match these characteristics, the LOPEN statement sets the port up correctly, and subsequent LPRINT statements will print there successfully.
- If your printer **does not** match these characteristics, you must use a procedure like the one listed below to change the characteristics of the serial port, before LPRINTs will print successfully to your printer.

Printers very often use DSR (DSR/DTR) handshaking, and you may need to set the port to use this.

SETTING THE SERIAL PORT CHARACTERISTICS

Calling the procedure

The `rsset:` procedure listed below provides a convenient way to set up the serial port.

Each time you use an `LOPEN “TTY:”` statement, follow it with a call to the `rsset:` procedure. Otherwise the LOPEN will use the default characteristics.

Passing values to the procedure

Pass the procedure the values for the five port characteristics, like this:

```
rsset:(baud%,parity%,data%,stop%,hand%,&0)
```

The final parameter, which should be `&0` here, is only used when reading from the port.

To find the value you need for each characteristic, use the tables below. You **must** give values to all five parameters, in the correct order.

Baud	50	75	110	134	150	300	600	1200	1800	2000	2400	3600	4800	7200	9600	19200
value	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Parity	NONE	EVEN	ODD
value	0	1	2

Data bits	5, 6, 7 or 8
Stop bits	2 or 1

Handshaking	ALL	NONE	XON	RTS	XON+RTS	DSR	XON+DSR	RTS+DSR
value	11	4	7	0	3	12	15	8

THE `rsset:` PROCEDURE:

```
PROC rsset:(baud%,parity%,data%,stop%,hand%,term&)
  LOCAL frame%,srchar%(6),dummy%,err%
  frame%=data%-5
  IF stop%=2 :frame%=frame% OR 16 :ENDIF
  IF parity% :frame%=frame% OR 32 :ENDIF
  srchar%(1)=baud% OR (baud%*256)
  srchar%(2)=frame% OR (parity%*256)
  srchar%(3)=(hand% AND 255) OR $1100
  srchar%(4)=$13
  POKEL ADDR(srchar%(5)),term&
  err%=IOW(-1,7,srchar%(1),dummy%)
  IF err% :RAISE err% :ENDIF
ENDP
```

Take care to type this program in exactly as it appears here.

EXAMPLE OF CALLING THE PROCEDURE

```
PROC test:
  PRINT “Testing port settings”
  LOPEN “TTY:A”
  LOADM “rsset”
  rsset:(8,0,8,1,0,&0)
  LPRINT “Port OK” :LPRINT
  PRINT “Finished” :GET
  LCLOSE
ENDP
```

`rsset:(8,0,8,1,0,&0)` sets 1200 Baud, no parity, 8 data bits, 1 stop bit, and RTS/CTS handshaking.

ADVANCED USE

The section of the `rsset:` procedure which actually sets the port is this:

```
srchar%(1)=baud% OR (baud%*256)
srchar%(2)=frame% OR (parity%*256)
srchar%(3)=(hand% AND 255) OR $1100
srchar%(4)=$13
POKEL ADDR(srchar%(5)),term&
err%=IOW(-1,7,srchar%(1),dummy%)
IF err% :RAISE err% :ENDIF
```

The elements of the array `srchar%` contain the values specifying the port characteristics. If you want to write a shorter procedure, you could work out what these values need to be for a particular setup you want, assign these values to the elements of the array, and then use the IOW function (followed by the error check) **exactly** as above.

READING FROM THE SERIAL PORT

If you need to read from the serial port, you must also pass a parameter specifying terminating mask for the read function. If `term&` is not supplied, the read operation terminates only after reading exactly the number of bytes requested. In practice, however, you may not know exactly how many bytes to expect and you would therefore request a large maximum number of bytes. If the sender sends less than this number of bytes altogether, the read will never complete.

The extra parameter, `term&` , allows you to specify that one or more characters should be treated as terminating characters. The terminating character itself is read into your buffer too, allowing your program to act differently depending on its value.

The 32 bits of `term&` each represent the corresponding ASCII character that should terminate the read. This allows any of the ASCII characters 1 to 31 to terminate the read.

For example, to terminate the read when Control-Z (i.e. ASCII 26) is received, set bit 26 of `term&`. To terminate on Control-Z or `<CR>` or `<LF>` which allows text to be read a line at a time or until end of file set the bits 26, 10 and 13. In binary, this is:

0000 0100 0000 0000 0010 0100 0000 0000

Converting to a long integer gives &04002400 and this is the value to be passed in `term&` for this case.

Clearly `term&` cannot be used for binary data which may include a terminating character by chance. You can sometimes get around this problem by using `term&` and having the sender transmit a leading non-binary header specifying the exact number of full-binary data following. You could then reset the serial characteristics not to use `term&` , read the binary data, and so forth.

EXAMPLE READING FROM SERIAL PORT

This example assumes that each line sent has maximum length 255 characters and is terminated by a `<CR>` and that Control-Z signals the end of all the data.

```
PROC testread:
  LOCAL ret%,pbuf&,pbuf1&,buf$(255),end%,len%
  PRINT “Test reading from serial port”
  LOPEN “TTY:A”
  LOADM “rsset”                   REM receive at 2400 without h/shake
  rsset:(11,0,8,1,0,&04002000)   REM Control-Z or CR
  pbuf&=ADDR(buf$)                REM could be pbuf% on Series 3c
  DO                    REM read max 255 bytes, after leading count byte
     len%=255
     pbuf1&=pbuf&+1
     ret%=IOW(-1,1,#pbuf1&,len%)
     POKEB pbuf&,len%             REM len% = length actually read
     REM including terminator char
     end%=LOC(buf$,CHR$(26))      REM non-zero for Control-Z
     IF ret%<0 and ret%<>-43
          BEEP 3,500
          PRINT
          PRINT “Serial read error: “;ERR$(ret%)
     ENDIF
     IF ret%<>-43                 REM if received with terminator
          POKEB pbuf&,len%-1      REM remove terminator
          PRINT buf$              REM echo with CRLF
     ELSE
          PRINT buf$;              REM echo without CRLF
     ENDIF
  UNTIL end%
  PRINT “End of session” :PAUSE -30 :KEY
ENDP
```

Note that passing -1 as the first argument to I/O keywords means that the LOPEN handle is to be used. Also, OPL strings have a leading byte giving the length of the rest of the string, so the data is read beyond this byte. The byte is then poked to the length which was read.

PRINTING TO A FILE

PRINTING TO A FILE ON THE PSION

In your OPL program, specify the destination file with an LOPEN statement like this:

❺ `LOPEN “D:\PRINT\MEMO”`

❸ `LOPEN “B:\PRINT\MEMO.TXT”`

❸ PRINTING TO A FILE ON A PC OR APPLE MACINTOSH

As if you were going to transfer a file:

- Physically connect the Psion and the other computer.
- Select the ‘Remote link’ option in the System screen and press Enter.
- Run the server program (supplied with 3 Link if you are using a Series 3c or Siena, or with the Series 5 itself) on the other computer.

In your OPL program, specify the destination file with an LOPEN statement.

For example, to a PC:

```
LOPEN “REM::C:\BACKUP\PRINTOUT\MEMO.TXT”
```

Any subsequent LPRINT would go to the file `MEMO.TXT` in the directory `\BACKUP\PRINTOUT` on the PC’s drive `C:`.

With a Macintosh, you might use a file specification like this:

```
LOPEN “REM::HD40:MY BACKUP:PRINTED:MEMO5”
```

An LPRINT would now go to the file `MEMO5` in the `PRINTED` folder, itself in the `MY BACKUP` folder on the hard drive `HD40`. Note that colons are used to separate the various parts of the file specification.

APPENDIX D

CHARACTER CODES

THE CHARACTER SET FOR THE SERIES 5

The following are character code descriptions for the 8-bit character set used in the English version of EPOC32. The definition is intended to be synonymous with code page 1252 (also used by Microsoft for Windows systems).

Those items in the ‘character’ column which are given in italics are descriptions. Particularly note that the code for *backspace* will be the code returned when you press Del, that for *carriage return* will be returned when you press Enter and that for *escape* when you press Esc.

hex code	*dec. code*	*character*	*hex code*	*dec. code*	*character*
00	000	*null*	01	001	*start of heading*
02	002	*start of text*	03	003	*end of text*
04	004	*end of transmission*	05	005	*enquiry*
06	006	*acknowledge*	07	007	*bell*
08	008	*backspace*	09	009	*horizontal tabulation*
0A	010	*line feed*	0B	011	*vertical tabulation*
0C	012	*form feed*	0D	013	*carriage return*
0E	014	*shift out*	0F	015	*shift in*
10	016	*data link escape*	11	017	*device control one*
12	018	*device control two*	13	019	*device control three*
14	020	*device control four*	15	021	*negative acknowledge*
16	022	*synchronous idle*	17	023	*end of transmission block*
18	024	*cancel*	19	025	*end of medium*
1A	026	*substitute*	1B	027	*escape*
1C	028	*file separator*	1D	029	*group separator*
1E	030	*record separator*	1F	031	*unit separator*
20	032	*(space)*	21	033	!

hex code	*dec. code*	*character*	*hex code*	*dec. code*	*character*	*hex code*	*dec code*	*character*
22	034	“	23	035	#	24	036	$
25	037	%	26	038	&	27	039	‘
28	040	(	9	041	)	2A	042	*
2B	043	+	2C	044	,	2D	045	-
2E	046	.	2F	047	/	30	048	0
31	049	1	32	050	2	33	051	3
34	052	4	35	053	5	36	054	6

37	055	7	38	056	8	39	057	9
3A	058	:	3B	059	;	3C	060	<
3D	061	=	3E	062	>	3F	063	?
40	064	@	41	065	A	42	066	B
43	067	C	44	068	D	45	069	E
46	070	F	47	071	G	48	072	H
49	073	I	4A	074	J	4B	075	K
4C	076	L	4D	077	M	4E	078	N
4F	079	O	50	080	P	51	081	Q
52	082	R	53	083	S	54	084	T
55	085	U	56	086	V	57	087	W
58	088	X	59	089	Y	5A	090	Z
5B	091	[	5C	092	/	5D	093	]
5E	094	^	5F	095	_	60	096	‘
61	097	a	62	098	b	63	099	c
64	100	d	65	101	e	66	102	f
67	103	g	68	104	h	69	105	i
6A	106	j	6B	107	k	6C	108	l
6D	109	m	6E	110	n	6F	111	o
70	112	p	71	113	q	72	114	r
73	115	s	74	116	t	75	117	u
76	118	v	77	119	w	78	120	x
79	121	y	7A	122	z	7B	123	{
7C	124	|	7D	125	}	7E	126	~
7F	127	*delete*	80	128	*not used*	81	129	*not used*
82	130	‚	83	131	ƒ	84	132	„
85	133	…	86	134	†	87	135	‡
88	136	ˆ	89	137	‰	8A	138	Š
8B	139	‹	8C	140	Œ	8D	141	*not used*
8E	142	*not used*	8F	143	*not used*	90	144	*not used*
91	145	‘	92	146	’	93	147	“
94	148	”	95	149	•	96	150	–
97	151	—	98	152	˜	99	153	™
9A	154	š	9B	155	›	9C	156	œ

9D	157	*not used*	9E	158	*not used*	9F	159	Ÿ
A0	160	*no-break space*	A1	161	¡	A2	162	¢
A3	163	£	A4	164	¤	A5	165	¥
A6	166	¦	A7	167	§	A8	168	¨
A9	169	©	AA	170	ª	AB	171	«
AC	172	¬	AD	173	-	AE	174	®
AF	175	¯	B0	176	°	B1	177	±
B2	178	²	B3	179	³	B4	180	´
B5	181	µ	B6	182	¶	B7	183	·
B8	184	¸	B9	185	¹	BA	186	º
BB	187	»	BC	188	¼	BD	189	½
BE	190	¾	BF	191	¿	C0	192	À
C1	193	Á	C2	194	Â	C3	195	Ã
C4	196	Ä	C5	197	Å	C6	198	Æ
C7	199	Ç	C8	200	È	C9	201	É
CA	202	Ê	CB	203	Ë	CC	204	Ì
CD	205	Í	CE	206	Î	CF	207	Ï
D0	208	Ð	D1	209	Ñ	D2	210	Ò
D3	211	Ó	D4	212	Ô	D5	213	Õ
D6	214	Ö	D7	215	×	D8	216	Ø
D9	217	Ù	DA	218	Ú	DB	219	Û
DC	220	Ü	DD	221	Ý	DE	222	Þ
DF	223	ß	E0	224	à	E1	225	á
E2	226	â	E3	227	ã	E4	228	ä
E5	229	å	E6	230	æ	E7	231	ç
E8	232	è	E9	233	é	EA	234	ê
EB	235	ë	EC	236	ì	ED	237	í
EE	238	î	EF	239	ï	F0	240	ð
F1	241	ñ	F2	242	ò	F3	243	ó
F4	244	ô	F5	245	õ	F6	246	ö
F7	247	÷	F8	248	ø	F9	249	ù
FA	250	ú	FB	251	û	FC	252	ü
FD	253	ý	FE	254	þ	FF	255	ÿ

ENTERING CHARACTERS AT THE KEYBOARD

Any of the characters in the character set can be entered directly from the keyboard:

1. Look up the **decimal** code of the character you want from the preceding table.
2. Hold down the Ctrl key and type the three-digit code, then release the control key.

Make sure that you include any preceding zeros to make the code three digits long - for example use Ctrl+096 to enter a single left quote, ‘.

MODIFICATION OF CHARACTER CODES

The keycode value is modified when pressing Ctrl at the same time as another key.

For alphabetic keys the Ctrl modified value is the ordinal position in the alphabet ignoring upper and lower case, i.e. 1 for Ctrl+A or Shift+Ctrl+A, 2 for Ctrl+B or Shift+Ctrl+B, etc. This value can be found by ANDing the ASCII value of the alphabetic character with (NOT $60). So the keycode of upper and lower case letters is the same when pressed together with Ctrl, with only the value of the modifier differing. For example, pressing Ctrl+J returns 10 which is (`%j AND (NOT $60)`).

OTHER SPECIAL KEYS

The keycodes (returned by GETEVENT32, etc) for these special keys are as follows:

key	*hex*	*decimal*	*key*	*hex*	*decimal*
Home	1002	4098	End	1003	4099
PgUp	1004	4100	PgDn	1005	4101
◀	1007	4103	▶	1008	4104
▲	1009	4105	▼	100A	4106
Menu	1036	4150			

GET and KEY, however, return the same values as on the Series 3c (see below).

CHARACTER CODES ON THE SERIES 3C AND SIENA

To find out a character’s character code either look up the character in the table given in the back of your User Guide, or press the Calc button and type the `%` sign followed by the character for example `%P` returns 80. Characters with codes from 0 to 127 are the same as in the ASCII character set. Codes 128 to 255 are compatible with the IBM code page 850.

Codes from 256 upwards are for other Series 3c keys see the list below.

CHARACTER CODES OF SPECIAL KEYS

The GET and KEY functions return the character code of the key that was pressed. Some of the keys are not in the character set. They return these numbers:

Esc	27	Tab	9
Delete	8	Enter	13
↑	256	↓	257
→	258	←	259

Pg Up	260	Pg Dn	261
Home	262	End	263
Menu	290	Help	291
⊞	292		

The Psion key adds 512 to the value of the key pressed. For example, Psion-a is 609 (512+97), and Psion-Help (Dial) is 803 (512+291).

SPECIAL CHARACTER CODES WITH PRINT

These values can be used with PRINT and CHR$():

7	beep
8	backspace
9	tab
10	line feed
12	form feed (clear screen)
13	carriage return (cursor to left of window)

For example, `PRINT CHR$(8)` moves the cursor backwards, one character to the left.

KEYBOARD MODIFIERS

Keyboard modifier values are the same for all machines, except that the Psion key (modifier value 8) does not exist on the Series 5.

`GETEVENT a%()` returns the modifier for a keypress in `(a%(2) AND $ff)`.

❺ `GETEVENT32 ev&()` returns modifiers to `ev&(4)` for keypresses and in `ev&(5)` for pointer (pen) events.

The values which can be returned are as follows:

modifier	*value*	*constant name (Series 5 only)*
Shift	2	`KKmodShift%`
Control	4	`KKmodControl%`
Caps	16	`KKmodCaps%`

❺ For the Series 5 there is additionally a Fn key:

Fn	32	`KKmodFn%`

❸ For the Series 3c there is additionally a Psion key:

Psion	8	-

APPENDIX E

LISTING OF CONST.OPH FOR SERIES 5

```
REM CONST.OPH version 1.10
REM Constants for OPL
REM Last edited on 19 May 1997
REM (C) Copyright Psion PLC 1997

REM General constants
CONST KTrue%=-1
CONST KFalse%=0

REM Data type ranges
CONST KMaxStringLen%=255
CONST KMaxFloat=1.7976931348623157E+308
CONST KMinFloat=2.2250738585072015E-308
     REM Minimum with full precision in mantissa
CONST KMinFloatDenorm=5e-324
     REM Denormalised (just one bit of precision left)
CONST KMinInt%=$8000
     REM -32768 (translator needs hex for maximum ints)
CONST KMaxInt%=32767
CONST KMinLong&=&80000000   REM -2147483648 (hex for translator)
CONST KMaxLong&=2147483647

REM Special keys
CONST KKeyEsc%=27
CONST KKeySpace%=32
CONST KKeyDel%=8
CONST KKeyTab%=9
CONST KKeyEnter%=13
CONST KGetMenu%=4150
CONST KKeyUpArrow%=4105
CONST KKeyDownArrow%=4106
CONST KKeyLeftArrow%=4103
CONST KKeyRightArrow%=4104
CONST KKeyPageUp%=4100
CONST KKeyPageDown%=4101
CONST KKeyPageLeft%=4098
CONST KKeyPageRight%=4099

REM Month numbers
CONST KJanuary%=1
CONST KFebruary%=2
CONST KMarch%=3
CONST KApril%=4
CONST KMay%=5
CONST KJune%=6
CONST KJuly%=7
CONST KAugust%=8
CONST KSeptember%=9
CONST KOctober%=10
CONST KNovember%=11
CONST KDecember%=12
```

```
REM Graphics
CONST KDefaultWin%=1
CONST KgModeSet%=0
CONST KgModeClear%=1
CONST KgModeInvert%=2
CONST KtModeSet%=0
CONST KtModeClear%=1
CONST KtModeInvert%=2
CONST KtModeReplace%=3

CONST KgStyleNormal%=0
CONST KgStyleBold%=1
CONST KgStyleUnder%=2
CONST KgStyleInverse%=4
CONST KgStyleDoubleHeight%=8
CONST KgStyleMonoFont%=16
CONST KgStyleItalic%=32

REM For 32-bit status words IOWAIT and IOWAITSTAT32
REM Use KErrFilePending% (-46) for 16-bit status words
CONST KStatusPending32&=&80000001

REM Error codes
CONST KErrGenFail%=-1
CONST KErrInvalidArgs%=-2
CONST KErrOs%=-3
CONST KErrNotSupported%=-4
CONST KErrUnderflow%=-5
CONST KErrOverflow%=-6
CONST KErrOutOfRange%=-7
CONST KErrDivideByZero%=-8
CONST KErrInUse%=-9
CONST KErrNoMemory%=-10
CONST KErrNoSegments%=-11
CONST KErrNoSemaphore%=-12
CONST KErrNoProcess%=-13
CONST KErrAlreadyOpen%=-14
CONST KErrNotOpen%=-15
CONST KErrImage%=-16
CONST KErrNoReceiver%=-17
CONST KErrNoDevices%=-18
CONST KErrNoFileSystem%=-19
CONST KErrFailedToStart%=-20
CONST KErrFontNotLoaded%=-21
CONST KErrTooWide%=-22
CONST KErrTooManyItems%=-23
CONST KErrBatLowSound%=-24
CONST KErrBatLowFlash%=-25
CONST KErrExists%=-32
CONST KErrNotExists%=-33
CONST KErrWrite%=-34
CONST KErrRead%=-35
CONST KErrEof%=-36
CONST KErrFull%=-37
CONST KErrName%=-38
CONST KErrAccess%=-39
CONST KErrLocked%=-40
CONST KErrDevNotExist%=-41
CONST KErrDir%=-42
CONST KErrRecord%=-43
CONST KErrReadOnly%=-44
CONST KErrInvalidIO%=-45
CONST KErrFilePending%=-46
CONST KErrVolume%=-47
CONST KErrIOCancelled%=-48
REM OPL specific errors
CONST KErrSyntax%=-77
CONST KOplStructure%=-85
CONST KErrIllegal%=-96
CONST KErrNumArg%=-97
CONST KErrUndef%=-98
CONST KErrNoProc%=-99
CONST KErrNoFld%=-100
CONST KErrOpen%=-101
CONST KErrClosed%=-102
CONST KErrRecSize%=-103
CONST KErrModLoad%=-104
CONST KErrMaxLoad%=-105
CONST KErrNoMod%=-106
CONST KErrNewVer%=-107
CONST KErrModNotLoaded%=-108
CONST KErrBadFileType%=-109
CONST KErrTypeViol%=-110
CONST KErrSubs%=-111
CONST KErrStrTooLong%=-112
CONST KErrDevOpen%=-113
CONST KErrEsc%=-114
CONST KErrMaxDraw%=-117
CONST KErrDrawNotOpen%=-118
CONST KErrInvalidWindow%=-119
CONST KErrScreenDenied%=-120
CONST KErrOpxNotFound%=-121
CONST KErrOpxVersion%=-122
CONST KErrOpxProcNotFound%=-123
CONST KErrStopInCallback%=-124
CONST KErrIncompUpdateMode%=-125
CONST KErrInTransaction%=-126

REM For ALERT
CONST KAlertEsc%=1
CONST KAlertEnter%=2
CONST KAlertSpace%=3
```

```
REM For BUSY and GIPRINT
CONST KBusyTopLeft%=0
CONST KBusyBottomLeft%=1
CONST KBusyTopRight%=2
CONST KBusyBottomRight%=3
CONST KBusyMaxText%=80

REM For CMD$
CONST KCmdAppName%=1    REM Full path name used to start program
CONST KCmdUsedFile%=2
CONST KCmdLetter%=3
REM For CMD$(3)
CONST KCmdLetterCreate$=”C”
CONST KCmdLetterOpen$=”O”
CONST KCmdLetterRun$=”R”

REM For CURSOR
CONST KCursorTypeNotFlashing%=2
CONST KCursorTypeGrey%=4

REM For DATIM$ - offsets
CONST KDatimOffDayName%=1
CONST KDatimOffDay%=5
CONST KDatimOffMonth%=8
CONST KDatimOffYear%=12
CONST KDatimOffHour%=17
CONST KDatimOffMinute%=20
CONST KDatimOffSecond%=23

REM For dBUTTON
CONST KDButtonNoLabel%=$100
CONST KDButtonPlainKey%=$200
CONST KDButtonDel%=8
CONST KDButtonTab%=9
CONST KDButtonEnter%=13
CONST KDButtonEsc%=27
CONST KDButtonSpace%=32

REM For dEDITMULTI and printing
CONST KParagraphDelimiter%=$06
CONST KLineBreak%=$07
CONST KPageBreak%=$08
CONST KTabCharacter%=$09
CONST KNonBreakingTab%=$0a
CONST KNonBreakingHyphen%=$0b
CONST KPotentialHyphen%=$0c
CONST KNonBreakingSpace%=$10
CONST KPictureCharacter%=$0e
CONST KVisibleSpaceCharacter%=$0f
```

```
REM For DEFAULTWIN
CONST KDefWin4ColourMode%=1
CONST KDefWin16ColourMode%=2

REM For dFILE
CONST KDFileNameLen%=255
      REM flags
CONST KDFileEditBox%=$0001
CONST KDFileAllowFolders%=$0002
CONST KDFileFoldersOnly%=$0004
CONST KDFileEditorDisallowExisting%=$0008
CONST KDFileEditorQueryExisting%=$0010
CONST KDFileAllowNullStrings%=$0020
CONST KDFileAllowWildCards%=$0080
CONST KDFileSelectorWithRom%=$0100
CONST KDFileSelectorWithSystem%=$0200

REM Opl-related Uids for dFILE
CONST KUidOplInterpreter&=268435575
CONST KUidOplApp&=268435572
CONST KUidOplDoc&=268435573
CONST KUidOPO&=268435571
CONST KUidOplFile&=268435594
CONST KUidOpxDll&=268435549

REM For DIALOG
CONST KDlgCancel%=0

REM For dINIT (flags for dialogs)
CONST KDlgButRight%=1
CONST KDlgNoTitle%=2
CONST KDlgFillScreen%=4
CONST KDlgNoDrag%=8
CONST KDlgDensePack%=16

REM For DOW
CONST KMonday%=1
CONST KTuesday%=2
CONST KWednesday%=3
CONST KThursday%=4
CONST KFriday%=5
CONST KSaturday%=6
CONST KSunday%=7

REM For dPOSITION
CONST KDPositionLeft%=-1
CONST KDPositionCentre%=0
CONST KDPositionRight%=1

REM For dTEXT
CONST KDTextLeft%=0
CONST KDTextRight%=1
CONST KDTextCentre%=2
CONST KDTextBold%=$100                      REM Ignored in Eikon
CONST KDTextLineBelow%=$200
CONST KDTextAllowSelection%=$400
CONST KDTextSeparator%=$800

REM For dTIME
CONST KDTimeAbsNoSecs%=0
CONST KDTimeAbsWithSecs%=1
CONST KDTimeDurationNoSecs%=2
CONST KDTimeDurationWithSecs%=3
REM Flags for dTIME (for ORing combinations)
CONST KDTimeWithSeconds%=1
CONST KDTimeDuration%=2
CONST KDTimeNoHours%=4
CONST KDTime24Hour%=8

REM For dXINPUT
CONST KDXInputMaxLen%=16

REM For FINDFIELD
CONST KFindCaseDependent%=16
CONST KFindBackwards%=0
CONST KFindForwards%=1
CONST KFindBackwardsFromEnd%=2
CONST KFindForwardsFromStart%=3

REM For FLAGS
CONST KFlagsAppFileBased%=1
CONST KFlagsAppIsHidden%=2

REM For gBORDER and gXBORDER
CONST KBordSglShadow%=1
CONST KBordSglGap%=2
CONST KBordDblShadow%=3
CONST KBordDblGap%=4
CONST KBordGapAllRound%=$100
CONST KBordRoundCorners%=$200
CONST KBordLosePixel%=$400

REM For gBUTTON
CONST KButtS3%=0
CONST KButtS3Raised%=0
CONST KButtS3Pressed%=1
CONST KButtS3a%=1
CONST KButtS3aRaised%=0
CONST KButtS3aSemiPressed%=1
CONST KButtS3aSunken%=2
CONST KButtS5%=2
CONST KButtS5Raised%=0
CONST KButtS5SemiPressed%=1
CONST KButtS5Sunken%=2

CONST KButtLayoutTextRightPictureLeft%=0
CONST KButtLayoutTextBottomPictureTop%=1
CONST KButtLayoutTextTopPictureBottom%=2
CONST KButtLayoutTextLeftPictureRight%=3
CONST KButtTextRight%=0
CONST KButtTextBottom%=1
CONST KButtTextTop%=2
CONST KButtTextLeft%=3
CONST KButtExcessShare%=$00
CONST KButtExcessToText%=$10
CONST KButtExcessToPicture%=$20

REM For gCLOCK
CONST KgClockS5System%=6
CONST KgClockS5Analog%=7
CONST KgClockS5Digital%=8
CONST KgClockS5LargeAnalog%=9
CONST KgClockS5Formatted%=11

REM For gCREATE
CONST KgCreateInvisible%=0
CONST KgCreateVisible%=1
CONST KgCreate2ColourMode%=$0000
CONST KgCreate4ColourMode%=$0001
CONST KgCreate16ColourMode%=$0002
CONST KgCreateHasShadow%=$0010

REM For GETCMD$
CONST KGetCmdLetterCreate$=”C”
CONST KGetCmdLetterOpen$=”O”
CONST KGetCmdLetterExit$=”X”
CONST KGetCmdLetterUnknown$=”U”

REM For gLOADBIT
CONST KgLoadBitReadOnly%=0
CONST KgLoadBitWriteable%=1

REM For gRANK
CONST KgRankForeground%=1
CONST KgRankBackGround%=32767

REM For gPOLY - array subscripts
CONST KgPolyAStartX%=1
CONST KgPolyAStartY%=2
CONST KgPolyANumPairs%=3
CONST KgPolyANumDx1%=4
CONST KgPolyANumDy1%=5
```

```
REM For gPRINTB
CONST KgPrintBRightAligned%=1
CONST KgPrintBLeftAligned%=2
CONST KgPrintBCentredAligned%=3
REM The defaults
CONST KgPrintBDefAligned%=KgPrintBLeftAligned%
CONST KgPrintBDefTop%=0
CONST KgPrintBDefBottom%=0
CONST KgPrintBDefMargin%=0

REM For gXBORDER
CONST KgXBorderS3Type%=0
CONST KgXBorderS3aType%=1
CONST KgXBorderS5Type%=2

REM For gXPRINT
CONST KgXPrintNormal%=0
CONST KgXPrintInverse%=1
CONST KgXPrintInverseRound%=2
CONST KgXPrintThinInverse%=3
CONST KgXPrintThinInverseRound%=4
CONST KgXPrintUnderlined%=5
CONST KgXPrintThinUnderlined%=6

REM For KMOD
CONST KKmodShift%=2
CONST KKmodControl%=4
CONST KKmodPsion%=8
CONST KKmodCaps%=16
CONST KKmodFn%=32

REM For mCARD and mCASC
CONST KMenuDimmed%=$1000
CONST KMenuSymbolOn%=$2000
CONST KMenuSymbolIndeterminate%=$4000
CONST KMenuCheckBox%=$0800
CONST KMenuOptionStart%=$0900
CONST KMenuOptionMiddle%=$0A00
CONST KMenuOptionEnd%=$0B00

REM For mPOPUP position type
REM Specifies which corner of the popup is given by the coordinates
CONST KMPopupPosTopLeft%=0
CONST KMPopupPosTopRight%=1
CONST KMPopupPosBottomLeft%=2
CONST KMPopupPosBottomRight%=3

REM For PARSE$ - array subscripts
CONST KParseAOffFSys%=1
CONST KParseAOffDev%=2
CONST KParseAOffPath%=3
CONST KParseAOffFilename%=4
CONST KParseAOffExt%=5
CONST KParseAOffWild%=6
REM Wild-card flags
CONST KParseWildNone%=0
CONST KParseWildFilename%=1
CONST KParseWildExt%=2
CONST KParseWildBoth%=3

REM For SCREENINFO - array subscripts
CONST KSInfoALeft%=1
CONST KSInfoATop%=2
CONST KSInfoAScrW%=3
CONST KSInfoAScrH%=4
CONST KSInfoAReserved1%=5
CONST KSInfoAFont%=6
CONST KSInfoAPixW%=7
CONST KSInfoAPixH%=8
CONST KSInfoAReserved2%=9
CONST KSInfoAReserved3%=10

REM For SETFLAGS
CONST KRestrictTo64K&=&0001
CONST KAutoCompact&=&0002
CONST KTwoDigitExponent&=&0004
CONST KSendSwitchOnMessage&=&010000

REM For GetEvent32
REM Array indexes
CONST KEvAType%=1
CONST KEvATime%=2

REM Event array keypress subscripts
CONST KEvAKMod%=4
CONST KEvAKRep%=5

REM Pointer event array subscripts
CONST KEvAPtrOplWindowId%=3
CONST KEvAPtrWindowId%=3
CONST KEvAPtrType%=4
CONST KEvAPtrModifiers%=5
CONST KEvAPtrPositionX%=6
CONST KEvAPtrPositionY%=7
CONST KEvAPtrScreenPosX%=8
CONST KEvAPtrScreenPosY%=9

REM Event types
CONST KEvNotKeyMask&=&400
CONST KEvFocusGained&=&401
CONST KEvFocusLost&=&402
CONST KEvSwitchOn&=&403
CONST KEvCommand&=&404
CONST KEvDateChanged&=&405
CONST KEvKeyDown&=&406
CONST KEvKeyUp&=&407
CONST KEvPtr&=&408
CONST KEvPtrEnter&=&409
CONST KEvPtrExit&=&40A

REM Pointer event types
CONST KEvPtrPenDown&=0
CONST KEvPtrPenUp&=1
CONST KEvPtrButton1Down&=KEvPtrPenDown&
CONST KEvPtrButton1Up&=KEvPtrPenUp&
CONST KEvPtrButton2Down&=2
CONST KEvPtrButton2Up&=3
CONST KEvPtrButton3Down&=4
CONST KEvPtrButton3Up&=5
CONST KEvPtrDrag&=6
CONST KEvPtrMove&=7
CONST KEvPtrButtonRepeat&=8
CONST KEvPtrSwitchOn&=9

CONST KKeyMenu%=4150
CONST KKeySidebarMenu%=10000

REM For PointerFilter
CONST KPointerFilterEnterExit%=$1
CONST KPointerFilterMove%=$2
CONST KPointerFilterDrag%=$4

REM Code page 1252 ellipsis (“windows latin 1”)
CONST KScreenEllipsis%=133
CONST KScreenLineFeed%=10

REM For gCLOCK
CONST KClockLocaleConformant%=6
CONST KClockSystemSetting%=KClockLocaleConformant%
CONST KClockAnalog%=7
CONST KClockDigital%=8
CONST KClockLargeAnalog%=9
REM GClock 10 no longer supported (use slightly changed gCLOCK 11)
CONST KClockFormattedDigital%=11

REM For gFONT (these may change before release)

CONST KFontArialBold8&=       268435951
CONST KFontArialBold11&=      268435952
CONST KFontArialBold13&=      268435953
CONST KFontArialNormal8&=     268435954
CONST KFontArialNormal11&=    268435955
CONST KFontArialNormal13&=    268435956
CONST KFontArialNormal15&=    268435957
CONST KFontArialNormal18&=    268435958
CONST KFontArialNormal22&=    268435959
CONST KFontArialNormal27&=    268435960
CONST KFontArialNormal32&=    268435961

CONST KFontTimesBold8&=       268435962
CONST KFontTimesBold11&=      268435963
CONST KFontTimesBold13&=      268435964
CONST KFontTimesNormal8&=     268435965
CONST KFontTimesNormal11&=    268435966
CONST KFontTimesNormal13&=    268435967
CONST KFontTimesNormal15&=    268435968
CONST KFontTimesNormal18&=    268435969
CONST KFontTimesNormal22&=    268435970
CONST KFontTimesNormal27&=    268435971
CONST KFontTimesNormal32&=    268435972

CONST KFontCourierBold8&=      268436062
CONST KFontCourierBold11&=     268436063
CONST KFontCourierBold13&=     268436064
CONST KFontCourierNormal8&=    268436065
CONST KFontCourierNormal11&=   268436066
CONST KFontCourierNormal13&=   268436067
CONST KFontCourierNormal15&=   268436068
CONST KFontCourierNormal18&=   268436069
CONST KFontCourierNormal22&=   268436070
CONST KFontCourierNormal27&=   268436071
CONST KFontCourierNormal32&=   268436072

CONST KFontCalc13n&=   268435493
CONST KFontCalc18n&=   268435494
CONST KFontCalc24n&=   268435495

CONST KFontMon18n&=    268435497
CONST KFontMon18b&=    268435498
CONST KFontMon9n&=     268435499
CONST KFontMon9b&=     268435500

CONST KFontTiny1&=     268435501
CONST KFontTiny2&=     268435502
CONST KFontTiny3&=     268435503
CONST KFontTiny4&=     268435504

CONST KFontEiksym15&=  268435661

CONST KFontSquashed&=  268435701
CONST KFontDigital35&= 268435752

REM For IOOPEN
REM Mode category 1
CONST KIoOpenModeOpen%=$0000
CONST KIoOpenModeCreate%=$0001
CONST KIoOpenModeReplace%=$0002
CONST KIoOpenModeAppend%=$0003
CONST KIoOpenModeUnique%=$0004

REM Mode category 2
CONST KIoOpenFormatBinary%=$0000
CONST KIoOpenFormatText%=$0020

REM Mode category 3
CONST KIoOpenAccessUpdate%=$0100
CONST KIoOpenAccessRandom%=$0200
CONST KIoOpenAccessShare%=$0400

REM Language code for CAPTION
CONST KLangEnglish%=1
CONST KLangFrench%=2
CONST KLangGerman%=3
CONST KLangSpanish%=4
CONST KLangItalian%=5
CONST KLangSwedish%=6
CONST KLangDanish%=7
CONST KLangNorwegian%=8
CONST KLangFinnish%=9
CONST KLangAmerican%=10
CONST KLangSwissFrench%=11
CONST KLangSwissGerman%=12
CONST KLangPortuguese%=13
CONST KLangTurkish%=14
CONST KLangIcelandic%=15
CONST KLangRussian%=16
CONST KLangHungarian%=17
CONST KLangDutch%=18
CONST KLangBelgianFlemish%=19
CONST KLangAustralian%=20
CONST KLangBelgianFrench%=21
CONST KLangAustrian%=22
CONST KLangNewZealand%=23
CONST KLangInternationalFrench%=24

REM End of Const.oph
```

APPENDIX F

SQL SPECIFICATION FOR THE SERIES 5

NOTE ON SYNTAX

The use of square brackets [ ] indicates that something is optional, while a vertical line | indicates a choice shold be made between mutually exclusive options. Words in heavy mono-spaced type (e.g. **SELECT**) should be typed in literally .

SQL keywords are case insensitive.

SELECT STATEMENT

select-statement :

SELECT *select-list*

FROM *table-name*

[**WHERE** *search-condition*]

[**ORDER BY** *sort-order*]

Use a *select-statement* to specify what data should be present in the view, and how to present it.

SELECTING COLUMNS

select-list :

*****

column-name-comma-list

Specify ***** to request that all columns for the table be returned in the view, in an undefined order; otherwise a comma separated list specifies which columns to return, and the order that the columns appear in the view.

NAMES

table-name

column-name

The table-name should be a table which exists in the database. Column names should refer to columns which exist in the specified table.

SEARCH CONDITION

search-condition :

boolean-term [**OR** *search-condition*]

boolean-term :

boolean-factor [**AND** *boolean-term*]

boolean-factor :

[**NOT**] *boolean-primary*

boolean-primary :

predicate

(*search-condition* **)**

This specifies a condition which a row must meet to be present in the generated view. A trivial search condition is just a single predicate, more complex search conditions are constructed by combining predicates using the keywords **AND**, **OR** and **NOT**, and using parentheses to override the standard precedence of these operators. Without brackets, the order of precedence is **NOT**, **AND** then **OR**. e.g.

```
a=1 or not b=2 and c=3
```

is equivalent to

```
(a=1 or ((not b=2) and c=3))
```

PREDICATES

predicate :

comparison-predicate
like-predicate
null-predicate

These are the building blocks of the search condition. Each predicate tests one condition of a column in the selected table.

COMPARISON PREDICATE

comparison-predicate :

column-name comparison-operator literal

comparison-operator :

< / > / <= / >= / = / <>

Compare a column value with a supplied literal value. Numeric columns (including bit columns) are compared numerically, text columns are compared lexically and date columns are compared historically. Binary columns cannot be compared. The literal must be of the same type (numeric, string, date) as the column.

LITERALS

literal :

string-literal
numeric-literal
date-literal

A *string-literal* is a character string enclosed in single quote characters ‘. To include a single literal quote character ‘ in a *string-literal*, use two literal quote characters ‘’.

A *numeric-literal* is any sequence of characters which can be interpreted as a valid decimal integral or floating point number.

A *date-literal* is a character string enclosed by the # character, which can be interpreted as a valid date.

LIKE PREDICATE

like-predicate :

column-name [**NOT**] **LIKE** *pattern-value*

pattern-value :

string-literal

Test whether or not a text column matches a pattern string. The wildcard characters used in the *pattern-value* are not standard SQL, instead the EPOC32 wildcard characters are used: ? for matching any single character and * for matching zero or more characters.

NULL PREDICATE

null-predicate :
    column-name **`IS`** `[` **`NOT`** `]` **`NULL`**

Test whether or not a column is Null. This predicate can be applied to all column types.

SPECIFYING A SORT ORDER

sort-order :
    sort-specification-comma-list

sort-specification :
    column-name `[` **`ASC`** `/` **`DESC`** `]`

Without an **`ORDER BY`** clause in the select statement the order that rows are presented is undefined. The columns specified in the sort-order can be ordered in ascending (the default) or descending order, and should appear in the sort-order in decreasing order of precedence. e.g.

```
surname, first_name
```

will order the rows by the column `surname`, and any rows with identical surnames will then be ordered by the column `first_name`.

APPENDIX G

EPOC32 ERROR VALUES

OPL

The error values returned by EPOC32 are listed below. Note that these are different from the error values returned by OPL in general. They may, however, be returned in the status word of asynchronous OPX functions.

meaning	*value*	*meaning*	*value*
no error	0	not found	-1
general error	-2	cancel	-3
no memory	-4	not supported	-5
argument error	-6	total loss of precision	-7
bad handle	-8	overflow error	-9
underflow error	-10	already exists	-11
path not found	-12	died	-13
in use	-14	server terminated	-15
server busy	-16	completion	-17
not ready	-18	unknown error	-19
corrupt	-20	access denied	-21
locked	-22	write error	-23
dismounted	-24	end of file	-25
disk full	-26	bad driver	-27
bad name	-28	comms line fail	-29
comms frame error	-30	comms overrun	-31
comms parity error	-32	timed out	-33
could not connect	-34	could not disconnect	-35
disconnected	-36	bad library entry point	-37
bad descriptor	-38	abort	-39
too big	-40	divide by zero	-41
bad power	-42	directory full	-43

INDEX