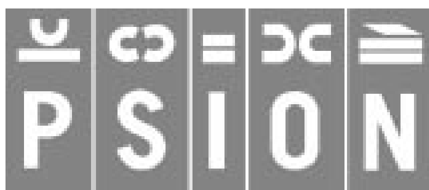
OPL

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.**



© Copyright Psion Computers PLC 1997

This manual is the copyrighted work of Psion Computers PLC, London, England.

The information in this document is subject to change without notice.

Psion and the Psion logo are registered trademarks. Psion Series 5, Psion Series 3c, Psion Series 3a, Psion Series 3 and Psion Siena are trademarks of Psion Computers PLC.

EPOC32 and the EPOC32 logo are registered trademarks of Psion Software PLC.

© Copyright Psion Software PLC 1997

All trademarks are acknowledged.

CONTENTS

CREATING & RUNNING PROGRAMS 1
CREATING A NEW MODULE 2
INSIDE THE PROGRAM EDITOR 2
AN EXAMPLE PROCEDURE TO TYPE IN 3
TRANSLATING A MODULE 4
RUNNING AFTER TRANSLATING 5
FILE MANAGEMENT 5
MORE ABOUT RUNNING MODULES 6
STOPPING A PROGRAM WHILE IT'S RUNNING 7
MENU OPTIONS WHILE EDITING 7
SUMMARY 9

VARIABLES & CONSTANTS 10
DECLARING VARIABLES 11
NUMBERS 11
TEXT 12
ARRAY VARIABLES 12
INITIAL VALUES 12
CHOOSING DESCRIPTIVE NAMES 12
GIVING VALUES TO VARIABLES 13
ASSIGNING VALUES 13
ARITHMETIC OPERATIONS 14
VALUES FROM FUNCTIONS 14
EXPRESSIONS 15
CONSTANTS 15
PROBLEMS WITH INTEGERS 16
OPERATIONS ON STRINGS 16
DISPLAYING VARIABLES 16
WHERE THE CURSOR GOES AFTER A PRINT 17
DISPLAYING A LIST OF THINGS 17
DISPLAYING THE QUOTE CHARACTER 17
SINGLE KEYPRESSES 19
EXAMPLE USING GET$ 19
MODIFIER KEYS 19
SUMMARY 20

LOOPS & BRANCHES 21
REPEATING INSTRUCTIONS (LOOPS) 22
DO...UNTIL 22
WHILE...ENDWH 22
EXAMPLE USING WHILE...ENDWH 23
"NESTING" LOOPS - THE 'TOO COMPLEX' MESSAGE 24
EXAMPLE USING IF 24
OR OPERATOR 24
EXAMPLE USING DO...UNTIL AND IF 24
FUNCTIONS AS ARGUMENTS TO OTHER FUNCTIONS 24
LOGICAL OPERATORS 25
JUMPING TO A DIFFERENT LINE 26
JUMPING OUT OF A LOOP: BREAK 26
JUMPING TO THE TEST CONDITION: CONTINUE 26
JUMPING TO A 'LABEL': GOTO 26
UNTIL 0, WHILE 1 27
SUMMARY 28

CALLING PROCEDURES 29
USING MORE THAN ONE PROCEDURE 30
MODULES CONTAINING MORE THAN ONE PROCEDURE 30
CALLING PROCEDURES 30
USES OF CALLING PROCEDURES 31
PARAMETERS 31
MULTIPLE PARAMETERS 32
RETURNING VALUES 33
GLOBAL VARIABLES 34
PASSING BACK VALUES 34
'UNDEFINED EXTERNALS' ERROR 35
SERIES 5 HEADER FILES, CONSTANTS AND PROCEDURE PROTOTYPES 35
SUMMARY 36

INDEX 37

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, *either*

- 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.

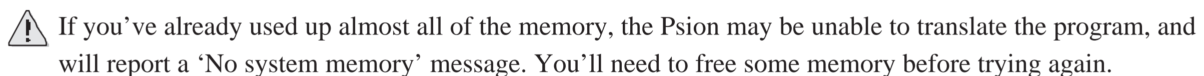
The Series 3c 'Prog' menu also has a 'S3 translate' option, for translating the current program in a form which can run on a Series 3 (as opposed to a Series 3c).

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

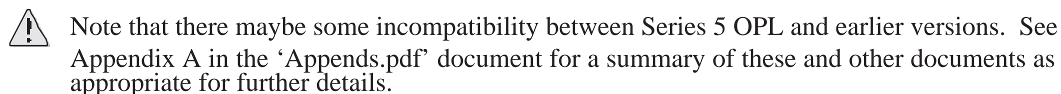
While you're typing in the procedure, all the options on the 'Edit' menu such as 'Copy' ('Copy text' on the Series 3c) and 'Paste' ('Insert text' on the Series 3c) - are available and can be used as in Word. Refer to the User Guide for more information.

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 `V234567890123456789012345678901 2%` is too long to be a valid variable name, but `V2345678901234567890123456789 01%` 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.)

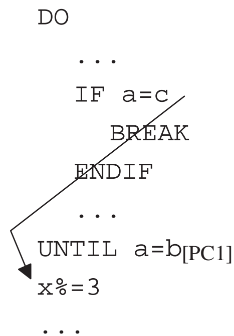
Example	*Result*	*Integer returned*
c1 AND c2	True if both c1 and c2 are true	-1
	False if either c1 or c2 are false	0
c1 OR c2	True if either c1 or c2 is true	-1
	False if both c1 and c2 are false	0
NOT c1	True if c1 is false	-1
	False if c1 is true	0

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

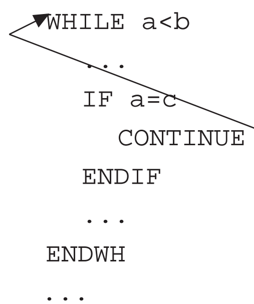
The BREAK command jumps out of a DO...UNTIL or WHILE...ENDWH structure. The line after the UNTIL or ENDWH statement is executed, and the lines following are then executed as normal. For example:



```
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.

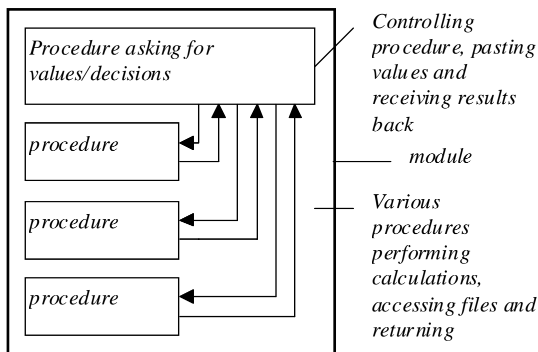
The top procedure is always the one which is executed, but it may also call, by name, any of the other procedures in the module. This procedure may in turn return a value, for example the result of a calculation, to the calling procedure.

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:



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

SYMBOLS

% operator 24
? prompt 18

A

arguments 15, 24
arithmetic operators 14
array variables 12
assign, value to variable 13
AT 18

B

bold text
  while editing 8
BREAK 26

C

calling procedures 30
case of OPL keywords 2
character codes
  with GET,GET$,KEY,KEY$ 19
coercion 35
commands
  and functions 14
conditional operators 25
conditions, in loops 22
CONST 36
Const.oph 36
constants 15, 35
CONTINUE 26
Control-Calc 6, 7
Control-S 7
Control-Shift-Calc 7
Control-Word 4, 7
copying modules 5
'Create standard files' option 8, 36
Ctrl+Esc 7
Ctrl+Fn+S 7

D

'Declaration' error 12
DECLARE EXTERNAL 35
declaring variables 11
  LOCAL and GLOBAL 34
'default' template 8
deleting modules 6
diamond
  key 8
division problems 16
DO...UNTIL 22
documents 5

E

ELSE 23
ELSEIF 23
ENDIF 23
ENDP 2
ENDV 27
Esc key, in INPUT, EDIT 18
'Export as text' option 8
expressions 15
EXTERNAL 35

F

false 25
'File is in use' 6
floating-point variables 11
  range 11, 12
'Format' menu 8
functions
  and commands 14

G

GET 19
GET$ 19
GLOBAL 34
Global variables
  returning values 34
global variables
  'Undefined externals' 35
'Go to' option 8
GOTO 26

H

header files 35
hexadecimal 16

I

IF...ENDIF 23
'Import text' option 8
INCLUDE 35
indentation 3, 8
'Infrared' option 8

initial values of variables 12
INPUT 15, 18
integer arithmetic 16
integer variables 11
  range 11

K

KEY 19
KEY$ 19
keypresses, recognising 19

L

labels 26, 27
LOCAL 11, 34
logical operators 25
logical values 25
long integer variables 11
  range 11
loops
  conditions 22
  DO...UNTIL 22
  IF...ENDIF 23
  maximum nested 24
  WHILE...ENDWH 22

M

modifiers 19
modules
  containing more than one procedure 30
  copying 5
  creating 2, 5
  deleting 6
  editing 3
  naming 2
  running 5, 6
  stopping while running 7
  translating 4

N

names of variables 12
'New file' option
  in Program editor 5
  in System screen 2, 5
'No system memory' 4, 5
non-document files 5
number input 18

O

'Open file' option
  in Program editor 5
operators
  arithmetic 14
  conditional 25
  logical 25
OR 24
'Outline' option 8
'Overflow' 16

P

parameters 31
  multiple 32
  'Type mismatch' 32
  types 32
Passwords
  on OPL programs 8
pausing a program 7
PRINT 14, 16, 18
PROC 2
procedure prototypes 35
procedures
  calling 30
  creating 2
  naming 3
  translating 4
'Prog' menu 4, 8
Program icon 5
proportional font
  while editing 8
Psion+Esc 7
Psion-Menu 7

R

range
  floating-point 11, 12
  integer 11
  long integer 11
REM 18
RETURN 33
returning values 33
'Run' option
  in Program editor 6
running a module 5, 6

S

'Save as' option 8
'Show error' option 8

‘Show last error’ option 8
statement 2
status window 7
STOP 27
stopping a running program 7
strings 12
  adding (concatenating) 16
  input 18
structures 22
‘Syntax error’ 4

T

tab width 8
template files
  in Program editor 8
text input 18
‘Too complex’ 24
‘Tools’ menu 8
‘Translate’ option 4
translating modules 4
true 25
‘Type mismatch’ 32
‘Type violation’ 35

U

UIDs
  application 5
‘Undefined externals’ 34, 35
UNTIL 0 27

V

variables
  array 12
  assigning values to 13
  declaring 11
  floating-point 11
  GLOBAL and LOCAL 34
  initial values 12
  integer 11
  long integer 11
  names of 12
  operations on 15
  string 12
  types 11
VECTOR 27

W

WHILE 1 27
WHILE...ENDWH 22