

# User's Guide

*for*

# Virtual Pascal

## v2.1

*containing*

Introduction

Installation

Virtual Pascal Features

Compatibility with Borland Pascal and Delphi

The VP User Interface

Writing programs using VP

Configuration and tools

Debugging programs with VP

VP example programs

Extending Virtual Pascal

Index

Copyright © 1996-1999 by *f*Print (UK) Ltd. All rights reserved.

All brand names and product names are trademarks or registered trademarks of their respective holders.

*f*Print (UK) Ltd  
Riverview House  
Beavor Lane  
London W6 9AR  
United Kingdom

Produced in the United Kingdom.

<b>CHAPTER 1 .....</b>	<b>6</b>
INTRODUCTION.....	6
<i>About this manual.....</i>	6
<i>What is Virtual Pascal?.....</i>	6
<i>History of development.....</i>	7
<i>System Requirements.....</i>	7
<i>Conventions used.....</i>	7
<i>Contacting Product support.....</i>	8
<i>Acknowledgements.....</i>	8
<b>CHAPTER 2 .....</b>	<b>10</b>
INSTALLATION.....	10
<i>Running the installation program.....</i>	10
<i>Component selection.....</i>	10
<i>Drive selection.....</i>	10
<i>Directory case.....</i>	10
<i>Installation directory.....</i>	11
<i>Source code and upgrades.....</i>	11
<i>Finishing the installation.....</i>	11
<i>Directory structure.....</i>	11
<b>CHAPTER 3 .....</b>	<b>13</b>
VIRTUAL PASCAL FEATURES.....	13
<i>The two OS/2 IDE variants.....</i>	13
<i>Key IDE Features.....</i>	13
The Debugger.....	14
<i>Key Compiler Features.....</i>	16
<i>The Run-Time Library.....</i>	17
<i>The built-in linker, VPLink.....</i>	18
<i>Program Examples.....</i>	19
<b>CHAPTER 4 .....</b>	<b>21</b>
COMPATIBILITY WITH BORLAND PASCAL AND DELPHI.....	21
<i>Run-Time Library compatibility.....</i>	21
<i>Data Type Compatibility.....</i>	22
Integers.....	22
Reals.....	23
Pointers.....	23
Classes.....	23
<i>Compiler directives.....</i>	24
<i>Built-in assembler (BASM).....</i>	24
<i>Inline statements and inline subroutines.....</i>	24
<i>Turbo Vision.....</i>	25
<i>Object Windows Library (OWL).....</i>	25
<i>Visual Component Library (VCL).....</i>	26
<i>Code generation.....</i>	26
<b>CHAPTER 5 .....</b>	<b>27</b>
THE VP USER INTERFACE.....	27
<i>Starting the VP IDE.....</i>	27
<i>Text-mode controls.....</i>	28
<i>File associations.....</i>	28
<i>Windows of the IDE.....</i>	29
Local Menus.....	29

## 4 Contents

---

The editor window: Colours .....	29
Cursor navigation .....	30
Editing .....	30
Block operations .....	31
Searching .....	31
Undo and Redo .....	32
Other editor options .....	32
Macros .....	32
Syntax Highlighting .....	33
Getting Help .....	33
The menu bar .....	33
The VP/PM Smartbar .....	35
Configuring the IDE .....	35
Options Resource compiler .....	36
Options Directories .....	36
Options Environment Preferences .....	37
Options Environment Editor .....	37
Options Environment Mouse .....	39
Options Environment Colours .....	39
Options Save options .....	39
<b>CHAPTER 6 .....</b>	<b>41</b>
WRITING PROGRAMS USING VP .....	41
<i>Writing your first program</i> .....	41
<i>Your second Virtual Pascal program</i> .....	42
<i>For more information</i> .....	42
<b>CHAPTER 7 .....</b>	<b>43</b>
CONFIGURATION AND TOOLS .....	43
<i>Changing settings in the IDE</i> .....	43
Options Compiler .....	43
Fastest setup .....	43
Smallest setup .....	43
Debug setup .....	43
The individual configuration options .....	43
Options Linker .....	49
Generating an import library .....	50
<i>Command-line tools</i> .....	51
The command line compiler VPC .....	51
Sample VPC response file .....	54
The Import Library manager IMPLIB .....	54
The Help Generator HG .....	55
The Help converter TPH2VPH .....	56
<i>Error Messages</i> .....	57
<b>CHAPTER 8 .....</b>	<b>58</b>
DEBUGGING PROGRAMS WITH VP .....	58
<i>Why debug?</i> .....	58
<i>Finding semantic errors</i> .....	58
Understanding compiler messages .....	59
A brief note on semicolons .....	59
Type conflicts .....	59
The compiler setup .....	60
<i>Preparing to debug</i> .....	61
Options Debugger .....	61
<i>Beginning the debug session</i> .....	62
Starting and stopping the program .....	62
Run-time errors .....	63

---

Run-time location information.....	63
Breakpoints.....	64
Exception trapping.....	65
Examining the state of the program.....	65
The Evaluator.....	66
The Watch Window.....	66
The Inspector.....	66
Format specifiers.....	67
Examples of format specifiers.....	68
Stepping through the program.....	69
Run Animate.....	69
Run Parameters.....	69
<i>Advanced debugging</i> .....	70
Advanced breakpoint options.....	70
The Call stack.....	70
The Log window.....	71
The Symbols window.....	71
The Unit list.....	71
The Threads list.....	71
The Object Hierarchy window.....	72
The Dump window.....	72
The Registers window.....	73
The NCP window.....	73
The CPU window.....	73
The CPU window as a stand-alone debugger.....	76
<b>CHAPTER 9</b> .....	<b>77</b>
VP EXAMPLE PROGRAMS.....	77
<i>Common Virtual Pascal examples</i> .....	77
Borland Pascal examples.....	78
<i>Windows-specific examples</i> .....	78
<i>OS/2-specific examples</i> .....	79
<b>CHAPTER 10</b> .....	<b>81</b>
EXTENDING VIRTUAL PASCAL.....	81
<i>Interface to the OS</i> .....	81
<i>RTL support</i> .....	81
<i>Linker support</i> .....	81
<i>Debugger support</i> .....	82
<i>Defining new targets in the IDE</i> .....	82
Restrictions in target definitions.....	84
Using the Command-Line Compiler.....	84
<i>Win32 Keyboard Support</i> .....	84
INDEX.....	86

# Introduction

## About this manual

Thank you for purchasing Virtual Pascal v2 for OS/2 and Windows. Throughout this manual, it will be referred to as **VP**.

This volume of the manual is intended for newcomers and experienced Pascal programmers alike. Users already familiar with Borland Pascal or Borland Delphi may skip over the chapters relating to the basics of writing a Pascal program and instead have a look at the list of unique Virtual Pascal features introduced in Chapter 3 on page 13.

If you just want to dive in and experiment, please refer to Chapter 2 for guidelines on installing VP on your computer. Once you have installed the product, have a look at the examples, or read Chapter 6 for an introduction to writing your own program.

All users can benefit from reading Chapter 7, which describes the intricacies of the built-in debugger.

For an in-depth description of the Pascal language implemented by VP and many technical details, please refer to the Language Reference Manual.

## What is Virtual Pascal?

VP is a cross-platform 32-bit compiler, linker and debugger, which supports both OS/2, Windows 95, Windows NT and certain compatible 32-bit DOS extenders. Its open architecture allows this compatibility to be extended to more Intel-based operating systems, including full source-level debugging.

VP is the best choice of compiler for any person or organisation desiring to produce fast, compact cross-platform applications as quickly as possible. Prior knowledge of the Pascal language is an advantage, but not a requirement. If you do not have any knowledge of the Pascal language, you should buy a book on Pascal programming to get the most out of Virtual Pascal - although the Language Reference Manual does provide enough information to get started, it is not intended as a textbook on the Pascal language.

VP features a very high level of compatibility with Borland Pascal 7.0 for DOS and Borland Delphi v1 to 3 for Windows. The primary aim of having a high degree of compatibility is to ensure that the process of porting existing code based on the DOS and Windows environments to native 32-bit OS/2 and Windows applications is as quick and smooth as possible. Being compatible also means that there already is a host of supplementary literature on the dialect of Pascal used by VP available - buy any Borland Pascal programming book and most of the issues apply to VP as well.

While being very compatible with the standards set by Borland, VP also offers a wide range of features unique to 32-bit development under OS/2 and Win32. All central parts of the system are prepared for multi-threading and the OS/2 and Win32 API functions are available and ready for use.

For new and experienced programmers alike, VP offers a productive, fast and modern development environment that enables users to make the best of development time.

## History of development

VP was initially developed and designed by Vitaly Miryanov while living in Ukraine and working at the Kiev Institute of Cybernetics. Many of the core parts of VP as it looks today were written on a PC AT-386DX-40Mhz with 4MB of RAM, as should be evident in the superb performance of both the IDE and the compiler itself. Although development has now moved to more modern computers, it is still an important part of the VP design philosophy to avoid today's multi-megabyte bloated applications that run slowly even on 400MHz Pentium IIs.

In June 1995, a public beta version of Virtual Pascal for OS/2 was released. Based on a thorough evaluation of the beta version, fPrint UK realised both the quality of the work already done and the big potential for a real, supported and properly documented version of VP.

fPrint UK contacted Vitaly who then became part of the development team at the fPrint offices in Hammersmith, London, where the initial versions of Virtual Pascal v1.x were developed.

In 1996, Virtual Pascal for OS/2 v1.0 was released, followed by an update to v1.1 later in the year.

In 1999, version 2 of Virtual Pascal was released, the first version supporting both OS/2 and Windows, implementing many changes to facilitate a true cross-platform system. Not only the IDE and compiler are cross-platform; the Run-Time Library included for your benefit allows you to create cross-platform programs as well.

With the takeover of fPrint by Peregrine Systems, Inc, in April 1999, development of Virtual Pascal virtually stopped, but – as this new v2.1 of Virtual Pascal shows – has been resumed.

Peregrine Systems has graciously agreed to sell all rights to Virtual Pascal to Allan Mertner, who has been lead developer of the project for the last few years, and now works on the project during spare time along with Vitaly Miryanov (who does the really difficult bits :)

We would like to take this opportunity to thank **Peregrine Systems** for making this possible! Peregrine Systems, Inc, can be contacted through <http://www.peregrine.com>.

## System Requirements

VP requires a PC with at least a 386 processor and 8MB of RAM, although more memory is recommended.

VP, including the IDE, debugger, compiler, linker and utilities, runs in OS/2 v2.x (with Workplace Shell) or later, Windows 95/98, Windows NT v4, and Windows 2000.

A Linux version is being worked on, which will support Linux with a kernel of 2.2 or later.

## Conventions used

This manual uses a consistent set of fonts and conventions making it easy and unambiguous reading. The conventions used are becoming standard in most literature of its kind and are outlined below:

### Courier

Is used to represent text as it appears in programs on the screen. Further, it is used for anything you must type on the OS/2 or Windows command line.

## 8 Introduction

---

### **bold**

A boldfaced font is used for Virtual Pascal reserved words, for command line options and compiler directives (**begin**, **{&Frame-}**, **/C**).

### KeyStroke

This font is used to represent a key or keys on your keyboard, for example “Press Ctrl-Esc to see the OS/2 task list or the Windows Start Menu”. Control character sequences, where you should hold down the Control key while pressing a character sequence is denoted with a leading caret (^). For example, ^KB means “Press Control and hold it down while pressing first K, then B”.

### [ ]

Square brackets are used to signify optional parameters to commands or other optional items.

### *Italics*

Are used when introducing new terms and to indicate Pascal identifiers. For a definition of an identifier, refer to the VP Language Reference.

### **File|New**

Is how menu item selections made in the VP IDE are displayed. **File|New** is equivalent to selecting the File menu (by pressing Alt-F or clicking on File on the menu bar) and selecting the New option from that menu.

## Contacting Product support

Support for VP is available either from your supplier or directly from the Virtual Pascal team. However, since Virtual Pascal is a spare-time project, we cannot teach you how to program Pascal – please refer to the help and manuals included and consult other literature available in the market.

Technical support is available by one of the following means:

### **World-Wide Web**

Virtual Pascal has its own web-site, which can be reached using a normal web-browser such as Netscape Navigator or Internet Explorer on the following address:

<http://www.vpascal.com>

The www-site contains up-to-date information about updates, bugs and new upcoming releases of Virtual Pascal.

### **e-mail**

e-mail is the preferred route for support requests. Please direct mails to [vpascal@vpascal.com](mailto:vpascal@vpascal.com); sales enquiries should be sent to [registrations@vpascal.com](mailto:registrations@vpascal.com).

### **Telephone support**

Telephone support is not available for Virtual Pascal.

## Acknowledgements

Virtual Pascal would never have become a reality without the support and enthusiasm offered by a large number of users worldwide. We appreciate the effort made by everybody and hope you find the product lives up to your expectations.



Special thanks goes - in no particular order - to

*Irina Miryanova, Kiev/London - for both patience and hard work*

*Christoph Bratschi, Switzerland - who did the VP/2 OWL clone*

*Anthony Busigin, Canada - for the MATHPAK library*

*Charlie Calvert, Borland US - for sending all those Delphi betas*

*Joanna Hodgson, EMEA DAP - for excellent technical support*

*Rick Timkovich, IBM US - for making the OS/2 API specs available*

*Matt Yurst, Canada - who wrote the VDUMP utility and LSXPower*

*Mr N Sriram, Singapore – for hosting the vpascal mailing list*

*Jörg Pleumann – for hard work on the Linux port*

*Veit Kannegieser – for making DPMI support available*

*Our spouses - for close to infinite patience*

*Dominos Pizza - for those late-night deliveries*

In addition, we would like to thank those hundreds of people who have patiently been sending bug reports, suggestions and feature requests.

*Thanks!*

# Installation

Virtual Pascal v2 is supplied with a text-mode installation program, one for OS/2 v2.0 or higher and one for Windows 95 and Windows NT/2000.

Before proceeding, unzip the Virtual Pascal archives downloaded to the same directory. This directory should not be the one to which VP is to be installed.

## Running the installation program

The Windows installation program is called W32Setup.Exe; the OS/2 one is called Os2Setup.Exe. Depending on the operating system on which you wish to install VP v2, execute one of the installation programs.

To do this, either navigate to the directory/folder containing the installation files using the Drives object (OS/2) or Explorer (Windows), or open a command line. Then, click on the installation program or type the name of the program to run, and press Enter.

When the installation program has started, it shows the Virtual Pascal licence agreement – please read it carefully before proceeding. Press Alt-I or Select File-Install to start the installation wizard.

On the first page of the installation wizard, click the “>>” button to proceed to the next page. At any stage during the installation setup procedure, click the button marked “<<” to return to the previous page.

## Component selection

On the main selection screen, select the VP options you wish to install. Some of the items contain sub-items, indicated in the status line along with the number of selected sub-components selected.

The check-box shows the current state of each item: a space indicates that no components from this group are selected; an X means that all components are selected, and a ? that some components are selected.

To include a component, click on the check box, or press Space. To select sub-components, click the Details button. Make the selection, and press the “>>” button.

When all desired components have been selected, click the >> button to continue to the drive selection dialog.

## Drive selection

This dialog shows a list of all available drive letters, their volume label and available disk space. Select the drive on which VP v2 should be installed. At the bottom of the screen, the disk space required for the selected options is displayed.

Then, click the >> button to proceed.

## Directory case

The installation program can create all directories of the VP installation either in lower case, UPPER CASE or Mixed Case. Select your preference in this dialog.

Then, click the >> button to proceed.

## Installation directory

By default, Virtual Pascal v2.1 is installed to the VP21 directory off the installation drive selected. To install to a different directory, please enter the directory name (without drive letter) in this dialog.

Click the >> button to complete the setup process or start the installation process.

## Source code and upgrades

Several components are supplied as upgrade patches, and are available only to users who own a licence of various Borland products: Turbo Vision, OWL and the Delphi VCL.

If any of these components were included in the Component Selection step, the installation program now prompts for the path to the original, unmodified, source files. The path required is usually the base directory of the original installation; the installation program automatically checks the relevant subdirectories.

For *Turbo Vision*, specify C:\BP\RTL (for example), and the installation program will look in the COMMON and TV sub-directories off this directory when looking for the original source files. If they are found, they will be copied to the Source\TV directory off the \VP21 directory, and upgraded to VP compatibility in place.

For *Object Windows Library (OWL)*, specify C:\BP\RTL (if upgrading from the Borland Pascal version), C:\DELPHI\SOURCE\RTL70 (if upgrading from the OWL code supplied with Delphi v1). The files are copied to Source\OWL and upgraded there.

For the *Visual Component Library (VCL)*, specify C:\DELPHI\SOURCE\VCL. Upgrade patches to upgrade the core of the VCL source code are included for Delphi v1.00, 2.00, 3.00 and 3.02. As for the other libraries, the original files are not modified but are copied to Source\VCL before they are upgraded to VP compatibility.

When all paths to existing source code have been entered, click >> to start the installation process.

## Finishing the installation

The installation program now unpacks and copies the files selected previously. At the end of the process, the installation program adds an item to the Start Menu (Windows) or the desktop (OS/2) and terminates.

VP is now ready to run – just open the Virtual Pascal object created, or run VP.EXE for your platform of choice.

The installation program does not modify the environment. In OS/2, it may be required to add the BIN.OS2 directory to the PATH and LIBPATH variables in CONFIG.SYS. In Windows, the PATH may be updated to point to the BIN.W32 directory.

Please refer to the following chapters for details about the IDE, debugger and command line utilities supplied with Virtual Pascal v2.1.

## Directory structure

The installation program creates a number of sub-directories; the hierarchy is explained below.

## 12 Installation

---

Each directory shown below is assumed to reside off the default main VP v2.1 installation directory (\VP21) and is not included. Some directories include %p. This indicates that two directories exist, one where %p is substituted for OS2 (containing OS/2-specific files), and one where %p is substituted for W32 (containing Win32-specific files).

This release also contains experimental Linux support files in LNX directories.

Directory	Contents
bin.%p	Binary (executable) files for each platform. This includes the compiler, the IDE, any command-line tools and any platform-specific help files.
lib.%p	Import libraries for each platform. The default libraries contain information about the system DLLs containing operating system APIs.
units.%p	Output directory for object files and other support files for each platform. Initially, these directories include precompiled versions of the Run-Time Library and Turbo Vision.
res.%p	Resource files for each platform.
out.%p	Executable output directory for each platform. Initially, these directories are empty.
out.%p\units	Output directory for each platform, where support files (.VPI, .LIB, .LNK, .ASM, etc.) are stored.
doc	Documentation (ReadMe, Language Reference, Users' Guide, licence agreement, etc.).
doc\os2	OS/2-specific documentation (API reference in .INF format)
common	Common files, such as online help, the vp.ini file used in both OS/2 and Win32, etc.
source	The root of the source files for various libraries.
source\%p	Platform-specific source code (Operating system API header files such as <i>Os2Base.Pas</i> )
source\rtl	Source for the Virtual Pascal Run-Time Library
source\rtl\sys	Source for the <i>System</i> unit of the Run-Time Library
source\tv	Source for the Turbo Vision library
source\owl	Source for the Object Windows Library
source\vcl	Source for the Visual Component Library
examples	The root of all example source code
examples\common	Example source code for all supported platforms
examples\%p	Platform-specific example code

## Virtual Pascal Features

Virtual Pascal is a professional quality 32-bit Object Pascal compiler for OS/2 and Win32. It includes both a command line compiler/linker and an Integrated Development Environment (IDE) with fully integrated debugger. VP can be used to develop both text mode and graphical applications and to port existing software products written in Borland Pascal or Borland Delphi to 32-bit OS/2 or Windows code.

This chapter is meant as an overview of the key features of Virtual Pascal. For descriptions on how to use a specific feature, please refer to the relevant sections in the Language Reference Manual.

Inspired by Borland Pascal and Turbo Debugger, the VP IDE is a text mode application. This means that the IDE does not look as flashy as some other modern development tools but also means that the resource requirements and speed of VP is unequalled.

### The two OS/2 IDE variants

For OS/2, VP offers two distinct versions of the powerful integrated development environment. The two environments look very much the same and share most capabilities, but serve two different purposes

The text-only version of the VP IDE runs on any computer running OS/2 v2.0 or later, even without the OS/2 Presentation Manager installed. It runs either in full screen mode or in any size window on the OS/2 desktop and offers the fastest possible operation. The drawback of this environment is the fact that OS/2's design, with only a single PM message queue, makes the writing and debugging of Presentation Manager applications difficult.

The PM version of the VP IDE was written to work around this design limitation of OS/2, allowing you to run and debug Presentation Manager applications without fear of crashing the entire system. In addition to this, VP/PM offers a graphical speedbar, whose smart-icons give quick access to the most commonly used features of the IDE with only a single click of the mouse. VP/PM requires OS/2 Presentation Manager and does not work if this is not present.

Apart from the differences noted above, the two IDE versions remain identical in both features and usage.

### Key IDE Features

- VP compiles in the background. While compilation is in progress, you can continue editing, browsing, read online help, etc.
- The editor allows multiple open windows, in each of which text files of any size can be edited. Special features worth mentioning include:
  - Unlimited number of Undo/Redo operations.
  - All block operations work with 3 block types: Line, Columnar and Stream.
  - Clipboard operations: Cut, Copy and Paste. VP/PM uses the OS/2 clipboard, making cut/paste operations between VP and other applications easy. The standard VP IDE for OS/2 and Win32 uses its own clipboard that is not accessible from the operating system.

## 14 Virtual Pascal Features

---

- Full Pascal syntax highlighting. Reserved words, numbers, text strings, comments and assembler statements are shown in their own configurable colour.
- Search and Replace with regular expressions.
- Incremental search facility.
- Mouse support. Right-clicking anywhere shows an intuitive pop-up menu offering quick access to some commonly used features.
- Full keystroke macro support, including a Macro List dialog box that displays currently defined keystroke macros and lets you add, remove and edit macros.
- Support for nested { } comments, up to 32 levels.
- Context-sensitive help system, including help for the IDE, the compiler, error messages, the Open Debug API and the Run-Time Library.
- If the help window is resized, the text is wrapped to fit the window.
- Topic search from editor and help windows.
- Search for text in the current topic, index or in all topics with Case Sensitive and Clipboard Copy and Copy Example commands.
- The text-mode IDE runs in a window and can have any size up to 250x50 characters. Set the desired screen size with the mode command and select **Options|Environment|Preferences|Keep Original** to make VP run without resizing the window.
- VP/PM allows dynamic sizing of the work area. When the window is resized, the editor windows are resized to fit the new size. The size and font is saved when VP/PM is exited and restored when it is next started (OS/2 only).
- The Integrated Debugger is a source level debugger that uses the Editor windows to facilitate source code tracking whilst executing.
- The IDE supports the special string **%P** in all directory and file name entries. Occurrences of **%P** are replaced by **OS2** when the compiler target is OS/2, and with **W32** when programming for a Win32 target. Using this feature, it is possible to use the same configuration file for cross-platform projects – simply change the target platform setting and recompile without making any further changes.
- All platform-specific files are stored in directories containing the 3 letters generated by **%P**. For example, OS/2 API header files are stored in Source\Os2, and those for Windows in Source\W32. OS/2 Import libraries are stored in Lib.Os2, and those for Windows in Lib.W32, meaning that the unit path can be set to Source\%P and the library path to Lib.%P, irrespective of target platform.

### The Debugger

- Sophisticated source tracing algorithm that allows complex lines to be stepped over at high speed (E.g: for I := 1 to 100000 do SomeFuncCall; )
- If the program being debugged fails with a run-time error or an exception, the source file containing the error is loaded and the cursor is positioned to the error location. If no source code is available, or debugging information is disabled, the CPU window is displayed instead, with the cursor moved to the offending instruction.
- Watch window showing the value of variables or even complex expressions. The window is automatically updated whenever necessary during program execution.
- Call stack window, showing the call stack, including parameter values.

- Breakpoints window. In addition to *execution breakpoints* it is possible to set up to four *datapoints* (supported by the CPU), each having a condition and a pass count for triggering it and an action that can be either Break or Log.
- Register window that shows the values of CPU registers and flags and allows them to be changed.
- Numeric Coprocessor window displaying the Stack registers, the Control register and the Status register panes showing the state of the x87 coprocessor as well as allowing you to change the state of it.
- Dump window that can show the memory in all Pascal integer and floating point formats.
- Object Hierarchy window, which shows an outline of your currently defined object hierarchy. Two different hierarchies are established; one for each object model supported by VP.
- Inspector window, which allows the inspection of complex structured variables in an efficient manner.
- Symbols window. Shows a list of all global and local symbols (types, constants, variables, procedures, functions) including their current values. The incremental search facility makes it easy to locate a specific symbol.
- CPU window with Code, Data, Register and Stack panes displays the source lines and the corresponding disassembled instructions. The code can be patched during execution using a built-in assembler similar to the one available in **asm.end** statements in a program.
- Log window collecting information about program flow (Module load, thread creation and termination, exceptions, breakpoints, user comments, etc). The Log window can also log expressions from Log-Points, which are similar to breakpoints, but log an expression instead of interrupting program execution.
- Evaluate/Modify dialog, used to evaluate expressions and modify the value of variables at run-time
- Messages window, containing information about compilation and linking progress.
- Threads dialog lets you examine all current program threads and allows you to freeze or resume them.
- Exceptions dialog which lists all OS/2 system exceptions and allows you to specify whether the debugger should handle a particular exception or if the user program is responsible for handling it. This can be particularly useful for catching hard-to-find bugs.
- Unit dialog list, listing all units used by the program along with include files. If a unit or an include file is selected, the source file is loaded into an editor window.
- The IDE can be used as a high speed assembler level debugger. Any 16- or 32-bit OS/2 executable or 32-bit Win32 executable can be loaded into the IDE using the **File|Load** Program menu item. The debugger can only load applications for the platform on which it is currently running.

### Key Compiler Features

- The compiler is very fast and generates efficient and small executables. On a low-spec machine (Pentium Pro-200), VP compiles and links roughly 600,000 lines of Pascal source code per minute.
- Generates optimised 32 bit code; including Pentium-specific optimisations.
- Borland Pascal version 7.0 compatible, including: Open array parameters, Open strings, Null-terminated strings, Dynamic methods and Small (integer-sized) sets.
- Borland Delphi v1.0 compatible. All language extensions introduced by Delphi v1.0 are available in VP in the **{&Delphi+}** state:
  - The class model (**class, property**),
  - Exception handling (**try..except, try..finally, raise**),
  - Guarded typecasts (**is, as**),
  - **Result** variable to assign and manipulate function result values,
  - Run-Time Type Information (RTTI),
  - Functions can return results of any type except file types,
  - Open Array constructors,
  - Method pointers,
  - Type variant open array parameters.
- Key 32-bit Delphi v2-v4 features are supported as well:
  - Local thread storage using **threadvars**,
  - Long strings, *AnsiString* and the **\$H** directive,
  - Initialised variables and read-only typed constants in **\$J+**.
  - Support for the *TMemoryManager* interface
  - The *Currency* floating point type is supported,
  - Unit **initialization** and **finalization** parts,
  - Double slash (*//*) comments,
  - The **assembler** directive is not required for assembler procedures and functions.
  - Separate output directory for .EXE and .MAP files, leaving support files (.OBJ, .LIB, .LNK, .VPI, etc) in a separate directory.
- Built-in Make and Build tools.
- 386/486/Pentium capable built-in assembler (BASM).
- VPC can generate object files, object libraries or readable assembly source that can be compiled by TASM (Turbo Assembler) 3.0+ or MASM (Microsoft Assembler) 6.0a+ to obtain equivalent object files. This allows you to mix code written in Virtual Pascal with code written in other languages.
- VPC supports the Pascal, C, StdCall and far16 Pascal (without thinking) calling conventions, allowing:
  - All OS/2 and Win32 API functions to be called directly without special interface libraries.
  - Operating system exception handlers etc to be written in Pascal.
  - 16-bit OS/2 1.x API functions to be called directly without any special interface thunk libraries.



- Direct, efficient I/O port and memory access through the *Port* and *Mem* arrays.
- Standard units are highly compatible with the DOS protected mode and Windows platforms of Borland Pascal.
- VP features Smart Linking, which lets you create tiny executables even without resorting to the use of DLLs. “Smart Linking” means that unused variables, procedures, functions and objects are not linked into the executable.
- VP can generate dynamic link libraries (DLL), usable from VP programs or programs written in another language, such as C or C++.
- Every definition in the interface section of a unit, including variables, typed constants, procedures, function and objects can be exported to a DLL and be used by any program using the DLL. This allows for the creation of DLLs even for complex object-oriented libraries like Turbo Vision and Object Professional. (OS/2 target only)
- No 64kB limits on either code segment, data segment or symbol sizes.
- The ampersand character (&) is allowed as a replacement for the dollar sign (\$) in compiler directives. This can be used to specify Virtual Pascal specific directives: other Pascal compilers, such as Borland Pascal, will treat these as ordinary comments.
- All Borland Pascal compiler directives are accepted. The ones that have no Virtual Pascal equivalents are ignored. \$A is equivalent to &AlignData.
- A number of Virtual Pascal specific compiler directives are supported: AlignCode, AlignData, AlignRec, Alters, Asm, Cdecl, Far16, Frame, G3, G4, G5, Open32, Optimise, OrgName, PureInt, Saves, SmartLink, Speed, StdCall, Use32, Uses, Zd.
- Compiler and IDE can be extended to support additional compiler targets, by modifying the VP.INI file located in the Common\ directory. For additional information on defining additional targets, please refer to the chapter Extending Virtual Pascal.

## The Run-Time Library

- The entire RTL is platform independent, with all platform-specific functions encapsulated in the *VpSysLow* unit. The *VpSysLow* unit, along with implementations for OS/2 and Win32 in *VpSysOs2.pas* and *VpSysW32.pas*, respectively, can be found in the Source\Rtl directory.
- The RTL includes the following units with full source code: *VpSysLow*, *System*, *Crt*, *Dos*, *Strings*, *TypeInfo*, *Use32*, *Rexx*, *VpUtils* and *WinDos*. Also included as pure interface units that can be updated to full source for owners of Delphi are *SysUtils*, *Classes* and *Math*. Owners of Turbo Vision can upgrade the source code to VP compatibility for *Objects*, *Memory*, *Validate*, and all the other TV units.
- Long file names are supported by the run-time library.
- The size of the file variables has been increased to hold a long file name. The *PathStr*, *DirStr*, *NameStr*, *ExtStr*, *SearchRec* types in the *Dos/WinDos* units have been modified accordingly.
- The section on Compatibility with Borland Pascal and Delphi contains a full explanation of compatibility and porting issues.
- In OS/2, a program (.EXE) or a library (.DLL) can use either the static or a dynamic version of any unit it uses. If using the static version, the unit’s object file is linked into the executable. If the dynamic version is used, the program requires the DLL to

## 18 Virtual Pascal Features

---

run. Full source level integrated debugging support is offered whichever method is used.

- For Win32, the low-level system wrapper functions (in *VpSysW32.Pas*) for keyboard handling can be replaced by similar functions placed in a DLL. This functionality has been included as a work-around for problems encountered on specific keyboard layouts. Please refer to the chapter on Extending Virtual Pascal for further information.
- Interface units providing access to almost all OS/2 base API calls are available. These are stored in the Source\os2 directory:
  - *Os2Base*: Base OS/2 API calls and 16-bit OS/2 1.x API (Vio, Kbd, Mou, Mon and Nls calls).
  - *Os2PmApi*: All Presentation Manager APIs.
  - *Os2MM*: All OS/2 Multimedia APIs.
  - *Os2Def*: Definitions of commonly used types and constants.
- Interface units providing access to almost all Win32 API calls are available. Most of these are stored in the Source\RTL directory, and some in the Source\W32 directory. (Since Open32 for OS/2 supports many of the Windows APIs, this unit is included in the common RTL directory):
  - *Windows*: Base Win32 API calls.
  - *Messages*: Message constants and message cracker types.
  - *CommDlg*: Common dialogs interface.
  - *WinSock*: Windows Sockets interface
  - *ShellApi*: Basic Win32 Shell API interface
  - *VpKbdW32*: Win32 Keyboard Handler code used by *VpSysLow*.

### The built-in linker, VPLink

VP includes a fast one-pass built-in linker, capable of producing both OS/2 LX and Win32 PE executables.

- Support for the **{*\$R ResourceFile*}** compiler directive. When this directive is specified, the linker automatically binds the resource files specified to the executable. Also supported is the form **{*\$R \*.RES*}**, which causes the resource file *ModuleName.Res* to be linked into the program.
- Compression (OS/2 only). OS/2 supports 2 levels of compression, Exepack 1 and Exepack 2. The VP linker supports both, and compresses both code and resources in accordance with the compression option selected in the Linker options dialog.
- Advanced *Stub* support. Any DOS program, even one using overlays, can be specified as a stub program. A stub program is a DOS program that is included in the executable and is loaded if the program is executed in DOS. The default stub is a small program (112 bytes) that runs the program if started from a Dos command prompt inside the native operating system.
- 4 levels of *Map files* can be automatically created: None, Segments, Public and Detailed. Map files are text files describing the internal layout of the executable, and can include information about segments, public symbols and line numbers. The map files produced by the VP linker are compatible with IBM's MAPSYM utility and can be used to create OS/2 .SYM files. They are also compatible with map files created using Borland's TLink.

- Support for an external linker. Note, that some features like location information are only available when using the built-in linker.
- New **&Linker** directive. Using this directive, an external .DEF file is not needed and the linker instructions can be kept inside the Pascal source files. If an external linker is used, the content of the **&Linker** directive will be copied to a .DEF file before invoking the linker.
- The built-in linker automatically recognises when direct port I/O is performed using the *Port* array and grants I/O privilege as required.
- Full support for resource files with string name identifiers, even in OS/2 (when using the Open32 library).
- The linker automatically generates an import library when linking a DLL, removing the need for the external ImpLib utility. When using an external linker, ImpLib is still used.
- For units and programs compiled in the **{&LocInfo+}** state, VPLink can generate Location Information in the executable. Location Information includes information about source file names and line numbers, which is accessible at run time through the *GetLocationInfo* function in the *System* unit and is used by the default error handlers in the *System* and *SysUtils* units to provide detailed error information on program failure.

## Program Examples

More than 80 examples of programming in VP are included in the package. Some are for OS/2 or Windows only, and others are for both platforms - for a discussion of the more interesting ones, please refer to Chapter 8.

- All Turbo Pascal examples can be compiled by Virtual Pascal with minor changes (BreakOut, TVDemo, TVEEdit, TVFM, etc.) The more interesting ones are included as OS/2 and Win32 executables. Since many versions of these examples exist, the Virtual Pascal installation includes .DIF files describing the changes required to each source file of the examples.
- Examples that show the use of the standard units *Dos*, *WinDos*, *Crt* and *WinCrt* are included both as source code and as executables.
- An example demonstrating the use of the *VPUtils* unit is included as *TestUtil\TestUtil.Pas*. The *VPUtils* unit is a Virtual Pascal specific part of the Run-Time Library.
- An example of converting C code to Pascal is included in the form of UNRAR. The original distributable C source code is included along with the working Pascal source.
- An example of assembler programming is included in the *Sort* example. This example implements a very fast routine for sorting text files and makes use of Virtual Pascal's built-in 32 bit assembler.
- A simple example of multithreading is included in the *RatRace* example. This example starts 8 threads, each of which displays a textual graphic (a "rat") on screen and races them across to the finish line.
- An example of the *TMemoryManager* class and performance monitoring can be found in the *MemMgr* directory.
- Two examples of interfacing to a Dynamic Link Libraries (DLLs) with VP can be found in the *Dll* directory. The source code for a simple DLL is also included.

## 20 Virtual Pascal Features

---

- Three examples of using the core of the Delphi VCL library, *Classes*, *SysUtils*, are included in the *Delphi* directory. *ClsDemo.Pas* demonstrates the *Classes* unit, *SysDemo.Pas* demonstrates the *SysUtils* unit, and *TlsDemo* demonstrates the use of Thread Local Storage (**threadvar** variables).
- Two OS/2 Presentation Manager programs, the *Triplex* game and the *Clock* application are included for OS/2 only.
- Two examples of 32-bit OWL (Object Windows Library) programming are included: *OwlChess* (a chess program) and *Calc* (a simple calculator). When compiling these, remember to define the `WINDOWS` and `OWL` conditional defines and do a Build of the project, rather than a Make.

## Compatibility with Borland Pascal and Delphi

Borland Pascal and Borland Delphi 1 generate 16 bit code whereas VP generates 32-bit code. The main differences between Borland Pascal/Delphi and VP are related to this fact. To achieve compatibility with these environments, VP uses a number of tricks and workarounds - they are described in this chapter.

If you do not plan to port Borland Pascal or Borland Delphi code to 32-bit OS/2 or Windows using VP, this chapter is probably of limited interest. In the following, a working knowledge of Borland Pascal or Borland Delphi is assumed.

### Run-Time Library compatibility

One of the most important aspects of compatibility is one of source code libraries and standard functions. This section is intended to give an overview of the changes made to the Run-Time Library units for VP.

- The following standard units are shipped with VP: *VpSysLow*, *System*, *SysUtils*, *Use32*, *Use16*, *Dos*, *Crt*, *Strings*, *Printers*, *WinDos*, *WinCrt*, *TypInfo*, *Math* and *Classes*. For the *SysUtils*, *Classes*, *Math* and *TypInfo* units, only the **interface** sections are included (because the original copyright belongs to Borland International); the other units are all supplied with full source code.
- The *VpSysLow* unit encapsulates all platform dependencies – no other units in the RTL (except *System*) use platform-specific features. This not only means that the code is not cluttered with IFDEFs, but also that it is possible, at least in theory, to make Virtual Pascal produce object code for other platforms by simply porting the *VpSysLow* unit. See chapter 10 about Extending Virtual Pascal for more information about this.
- The source for the *System* unit and a batch file to compile it (*MakeSys.Cmd*) can be found in the `Source\RTL\Sys` directory of the VP installation.
- All other common RTL units, including *VpSysLow*, can be found in the `Source\RTL` directory of the VP installation. A batch file for recompiling the entire RTL is also included (*BldRtl.Cmd*).
- All *Integer* and *Word* type parameters to standard procedures and functions as well as function return values of these types have been extended to *Longint*.
  - For this reason, the *Use32* unit should always be included in the `uses` clause before trying to recompile 16-bit code written with Borland Pascal. If it is not, variable parameters of types *Word* and *Integer* will cause *Type mismatch* errors when compiling.
  - To ease this process, the compiler can be instructed to implicitly include the *Use32* unit in the `uses` clause by specifying the `{&Use32+}` state.
  - When using `{&Use32+}` to include the *Use32* unit implicitly, it is possible to revert to the original 16-bit integer types by including the *Use16* unit in the `uses` clause. This unit simply reverts the changes done by the *Use32* unit.
  - Delphi units always use the *SysUtils* unit. The VP version of *SysUtils* redefines *Integer* as *Longint*, and the use of *Use32* is not required nor recommended for these units.
- The differences in each of the RTL units included consists of:

## 22 Compatibility with Borland Pascal

---

- *System* unit:
  - The *Mark* and *Release* real mode procedures are not implemented.
  - Since VP operates with a flat 32-bit memory model as opposed to a 16:16 bit segmented one, the *Ptr* function accepts only one argument, *Offset*.
  - Two new procedures for converting between 16:16 and 0:32 pointer types, *SelToFlat* and *FlatToSel* have been added.
  - Two new 16-bit integer types, *SmallInt* and *SmallWord* are introduced.
  - Three new variables supplementing *FileMode* are defined: *FileModeReadWrite*, which is used when using *ReWrite* on a typed or untyped file instead of *FileMode*, *TextModeRead*, used when *Reset* is used on a file of type *Text*, and *TextModeReadWrite*, used when *ReWrite* is used on a file of type *Text*. The file mode variables are all declared as **threadvar** and can safely be used in multi-threaded programs.
  - Thread and exception handling functions are added.
  - The *TMemoryManager* interface is added.
  - Several functions for handling of long strings are added.
  - A function for accessing run-time location information, *GetLocationInfo*, is added.
- *Crt* unit:
  - The *Sound* and *NoSound* procedures are not implemented; the VP procedure *PlaySound* implements a combination of both.
- *Dos*, *WinDos* units:
  - Procedures specific to the DOS operating system (*Intr*, *MsDos*, *GetCBreak*, *SetCBreak*, *Keep*, *GetIntVec* and *SetIntVec*) are not implemented.
  - *SwapVectors* is available, but does nothing.
- *WinCrt* and *Strings*: No changes.
- *Printers* unit:
  - In order to actually print what has been written to the *Lst* file, a call to the new *FlushPrinter* procedure must be made.
- *SysUtils* unit: No changes
- *TypInfo* unit: No changes
- *Classes* unit:
  - The *TThread* type is not supported.
- *Math* unit: No changes.

## Data Type Compatibility

By default, all built-in data types have the same name, size and meaning as they do in Borland Pascal. All data types of Borland Pascal and Delphi (except *Variant*) are supported and consist of *Byte*, *Word*, *Integer*, *LongInt*, *SmallInt*, *Cardinal*, *Currency*, *Real*, *Comp*, *Double*, *Extended*, *String*, *AnsiString* and *PChar*.

### Integers

On a 32-bit processor like the 80386 and later, it is not efficient to use 16 bit arithmetic.

For this reason, units should either use the *SysUtils* or the *Use32* unit, either of which should be included in the **uses** clause of every unit. *Use32* redefines the built-in 16-bit integer types *Integer* and *Word* as being 32-bit types and not much else, and *SysUtils* does

the same and a whole lot more. (The source code of *Use32* is available if you installed the RTL source code in the file `Source\Rtl\Use32.pas`).

As an alternative to manually changing the **uses** clause of all units, the `{&Use32+}` setting can be used, for example by setting the *Use32* option in the `Options\Compiler` dialog. When this directive is enabled, the *Use32* unit is implicitly included in the **uses** clause, after the *System* unit but before any units explicitly listed. Note, that this setting should be used for programs being ported from Borland Pascal only, not for programs that need to be compatible with any version of Borland Delphi.

If you must use 16-bit integer values, use the new *SmallInt* and *SmallWord* types instead of *Integer* and *Word*, or use the fully qualified identifiers: *System.Integer* and *System.Word*.

In VP, all *Byte* and *Word* sized operands are converted to type *Longint* for integer arithmetic operations.

See also the discussion on integer types in the `Data Types` chapter of the `Language Reference Manual`.

## Reals

For floating-point calculations, VP always uses coprocessor types internally (for reasons of speed and code size), but allows the use of 6-byte *Reals* for reasons of compatibility with older code. When operating on 6-byte *Reals*, these are converted to *Extended* and back again when the operation is complete. This obviously is a slow process and we recommend the use of *Reals* only when strictly necessary.

When using a variable of type *Real*, rounding errors can occur. These rounding errors are consistent with the results obtained from Borland Pascal programs using reals, compiled in the `{N+}` state.

## Pointers

Virtual Pascal uses a 32-bit flat memory model, making the use of segments a thing of the past. Pointers in Virtual Pascal have the same size as in Borland Pascal and Borland Delphi (4 bytes) but denote a 32-bit flat memory address rather than 16-bit segment (or selector) and 16-bit offset. Consequently, the *Mem* arrays have only one index - the 32-bit flat memory offset.

The same applies to the *Ptr* function, which also takes a single offset argument.

Some 16-bit OS/2 API functions, like *VioGetBuf*, returns segmented 16:16 bit pointers. In order to address the memory pointed to by this pointer, use the *SelToFlat* procedure defined in the *System* unit to convert the pointer to a flat 32-bit one and call *FlatToSel* to convert a normal 32-bit flat memory pointer to 16:16 for use with these functions.

## Classes

Classes and class variables are compatible with Delphi, from a compiler/language point of view. In other words, a class definition that compiles in Delphi will compile with Virtual Pascal, and vice versa.

However, it is not possible to create a DLL defining classes in one compiler, and use the DLL with a program compiled using the other. This is because the 32-bit Delphi compilers use the *Register* calling convention not supported by VP. Since the RTTI (Run-Time Type Information) stored for classes does not contain calling convention information, it is not possible to change the calling convention for read/write methods for properties, for example.

### Compiler directives

All Borland Pascal compiler directives implemented in VP have the same spelling. A number of new switch directives have been introduced in VP; these have descriptive names instead of being one-letter switches as used in Borland Pascal. A detailed description of all VP compiler directives can be found in the online help and in the Language Reference Manual.

Several Borland Pascal directives are not relevant in the 32-bit environments supported by VP and these are ignored:

Switch	Meaning
\$E	Emulation
\$F	Force FAR Calls
\$G	Generate 286 Instructions
\$K	Smart Callbacks
\$N	Numeric Coprocessor
\$O	Overlay Code Generation
\$Y	Symbol Reference Information

The following parameter directives are also ignored in VP:

Directive	Meaning
\$C Attribute	Code Segment Attribute
\$D Text	Description
\$G UnitName	Group Unit Segments
\$O UnitName	Overlay Unit Name
\$S SegSize	Segment Size Preference

### Built-in assembler (BASM)

The built-in assembler of VP supports all 386, 486 and Pentium instructions. Assembler code written for Borland Pascal or 16-bit Delphi should be rewritten to 32 bit, which usually is a very simple process. All segment register references can usually be discarded, since a single register can hold the offset of the desired memory location. As an example, **LDS BX,Test** loads the address of Test into DS:BX in 16-bit mode and should be replaced by **MOV EBX,Test** in 32-bit mode.

32-bit assembler code written in 32-bit Delphi should be converted to use the standard **pascal** calling convention, since VP does not support the **register** calling convention used by default in these compilers.

Two new operators are introduced: Small and Large. They should be used to modify the memory addressing mode (Small: 16-bit, Large: 32-bit). They have the same meaning and syntax as in other 32-bit assemblers, such as MASM or TASM.

Two pseudo instructions are available, ALIGN and POPARGS.

For a more complete discussion about the features of the built-in assembler, please refer to the Language Reference Manual.

### Inline statements and inline subroutines

The *Inline* procedure defined by Borland Pascal, where numbers representing machine code language can be entered directly, is not supported. This statement was introduced into Turbo Pascal prior to the introduction of the built-in assembler and should be avoided in Borland Pascal programs as well. Instead, use the built-in assembler for this kind of code - it is just as fast and much more readable.



VP supports inline subroutines written in Pascal. Inline subroutines are not called, but are instead expanded every time they are used, like macros. Declaring small functions that are used very often in the code as **inline** may give a speed improvement at the cost of code size. Larger functions should never be declared as **inline**.

For more information about *Inline* functions and restrictions imposed on them, please refer to the Language Reference Manual.

## Turbo Vision

VP includes a patch to make the Turbo Vision v2.0 source code compatible with it, for both OS/2 and Win32 targets. The *Drivers* and *Objects* unit have been completely rewritten and the assembler code of all other Turbo Vision units has been rewritten.

The DOS specific *TEmsStream* object is not implemented. The *TEvent* record has been extended to include a *ShiftState* byte field for the *evKeyDown* variant. Critical error handlers are not available, since OS/2 and Win32 handles critical errors properly. The *FormatStr* procedure has been rewritten in built-in assembler. High level units such as *Menus*, *Dialogs*, etc, remain virtually unchanged.

The original Turbo Vision v2.0 source code is supplied with Borland Pascal v7.0 and v7.01 in the package that includes the runtime library (RTL) source code, or the RTL can be purchased separately from Borland. In order to install source code support for Turbo Vision, the original Borland source code files must already be installed.

Users that do not have the Turbo Vision source code but wish to make use of the library with VP, the Turbo Vision library is included as precompiled units with Pure Interface units of all units in the Source\TV directory.

## Object Windows Library (OWL)

VP includes a patch with which the OWL library source code can be made VP-compatible. OWL support is available for Win32 targets only. The original source code for OWL (required to make use of this compatibility in Virtual Pascal) is included in Borland Pascal and Borland Delphi v1. Additionally, 32-bit OWL programs require the file BWCC32.DLL in order to run. This DLL is included with Borland C v4.x and higher, and should be widely available for download from the Internet.

Using the ported version of the OWL library source files, it is relatively easy to recompile 16-bit OWL applications written with Borland Pascal for Windows to 32-bit Windows.

Two examples of OWL compatibility (from Borland Pascal) are included as executables: CALC and OWLCHESS; for copyright reasons, the required BWCC32.DLL file is not included.

A few special considerations are warranted when compiling OWL: Define the conditional define **OWL**, and recompile the RTL, particularly including the *Objects* unit. In OWL, some of the functions are implemented differently than for Turbo Vision; the precompiled version of *Objects* is compiled for use with TV.

Also note, that while OWL programs may compile with no changes, the differences in messages between Win16 and Win32 may mean that the programs do not work, and some work will be required to find and remove the problems.

### Visual Component Library (VCL)

The basic parts of the Visual Component Library of Borland Delphi are included with Virtual Pascal: *TypInfo*, *SysUtils*, *Math* and *Classes*. The other units are not included with Virtual Pascal but most should be able to compile with only a few changes.

Using the Open32 library from IBM (Previously called DAX or DAPIE), it may be possible to cross-compile Delphi applications to an OS/2 target using Virtual Pascal.

To assist the porting process, VP defines a new compiler directive, **{&Open32+}**, which can only be used for OS/2 targets. In this state, all resource files included using a **\$R** directive are considered to be Win16 or Win32 resources and are converted to OS/2 resource format prior to linking.

Additionally, the meaning of the **stdcall** standard directive is changed to be identical to **cdecl** when the **&Open32** state is enabled. This is because the standard Windows calling convention is **stdcall**, and this is obviously used in all Windows API header files. However, IBM has implemented Open32 with the standard OS/2 calling convention **cdecl** – this directive allows the same Windows header files to be used for both Windows and OS/2 targets without changing the source code.

At the time of writing, Delphi programs ported to OS/2 using the Open32 library were both slow and visually unappealing. This may of course change in future versions of the Open32 library.

### Code generation

VP generally generates very efficient code. The main differences in code generation from Borland Pascal and Delphi are outlined here. This section is highly technical and only of interest for programmers with a thorough knowledge of Intel assembler programming.

- Nested procedures and functions use a different model for accessing local variables and arguments of an enclosing procedure or function based on the ENTER instruction.
- For overflow checking, VP uses the INTO instruction and the BOUND instruction to implement range checking. The result is that the overhead of enabling range and overflow checking is minimal, both in terms of code size and speed.
- VP replaces multiplications and divisions by powers of 2 by right- and left-shifts, speeding up execution.
- In the **{&Speed+}** state, VP replaces multiplication by a number of small constants (3, 5, 6, 9, etc) by equivalent LEA and shift instructions.
- VP always keeps a pointer to *Self* in a register for maximum performance when programming using objects. In the **{&Optimize+}** state, VP always attempts to store the most frequently used local variables in CPU registers instead of memory for optimum performance.
- VP generates very efficient **for** loop code. However, after the loop has terminated, the value of the FOR control variable is 1 greater than the maximum bound. Programs should not rely on the value of a **for** control variable outside the **for** loop.
- **Case** statements cause more efficient code to be generated. As opposed to Borland Pascal, VP does not allow overlapping case selectors.
- In the **&G5+** state (Pentium processor is the primary target), ENTER instructions are used instead of the usual *push ebp; mov ebp,esp* sequence. This also works on non-Pentium processors but is slightly slower on these.

## The VP User Interface

The IDE of VP is your most valuable tool in developing new programs and since you are likely to spend a lot of time using it, it is important that you learn to feel comfortable with it. If you have not already done so, you should spend a minute or two browsing the pull-down menus to get an overview of the available options. This will make it easier for you to follow and understand the information contained in this chapter.

### Starting the VP IDE

The IDE is launched like any other executable, either from the command-line or by double-clicking the icon on the desktop. If running OS/2, choose the IDE that suits your needs the best; a description of the differences between the two different IDEs (VP or VPPM) can be found on page 13.

Either IDE accepts a number of command-line parameters, which can be used to automatically load a source file or project on start-up. The start-up syntax is as follows:

```
VP [options] [files]
```

The following options are available for both IDEs:

Option	Description
/C<FileName>	Use configuration file <FileName>. If not specified, VP.VPO is used.
<FileName>	Load <FileName> into the editor
/S<Path>	For OS/2, specifies path of .SYM files
/H or /?	Display a help screen

If a configuration file is specified, all settings stored in the file will be restored on starting the IDE. This includes the compiler settings, desktop layout, editor windows, colour schemes and macros, etc. Alternatively, any number of source files can be specified and will be loaded into the editor. All parameters are optional.

When the IDE is not configured to use 25- or 50-line mode, but to keep the dimensions of the original window, the desired window size can be established by issuing a **Mode** command prior to invoking the VP IDE. This setting can be found in the Options!Environment!Preferences dialog.

VPPM (OS/2 only) supports an additional two parameters used to specify the desired font-size. By default, a font-size of 16x8 is selected and this can be overridden on the command-line. Once a new font-size has been selected, this font becomes the new default and does not have to be specified on the command line again.

Specify the font-size by specifying two extra parameters:

```
VPPM /Fx<SizeX> /Fy<SizeY>,
```

for example

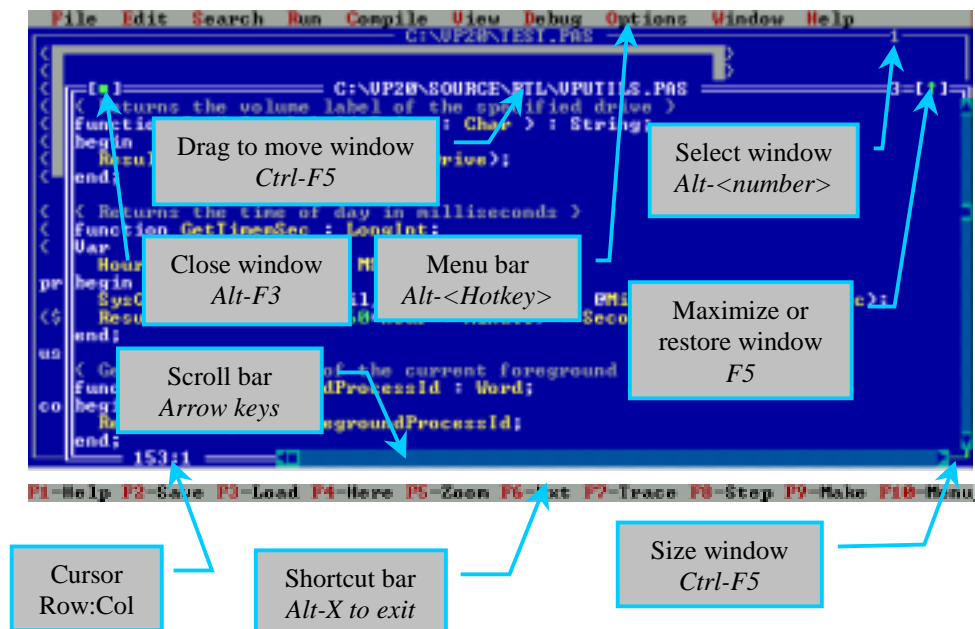
```
VPPM /Fx30 /Fy20.
```

This will select a 30x20 font, or the closest matching font available on your system.

In OS/2, the debugger supports .SYM files for debugging. Please refer to page 76 for more information.

## Text-mode controls

The currently active window in the IDE has a number of different icons on the border of the window, used to manipulate its location and size on the desktop. Below is a picture of a screenshot, showing commonly used functions and their keyboard shortcuts.



Each of these options can also be activated using the Window option located on the pulldown menu at the top of the screen.

In addition to the above icons, editor windows also display the current cursor position (as column:line number) at the bottom of the screen. Windows may also contain scroll bars, which can be used for navigating through large files.

## File associations

Both OS/2 and Windows can use the extension of data files to associate groups of files with an application.

During installation, VP creates an association between files with a **.PAS** extension and the VP IDE, after which double clicking any **.PAS** file automatically loads VP with the selected file.

To change the association to another program, different methods are required in Windows and OS/2:

OS/2: Open the settings notebook for the application you wish to associate **.PAS** file to. Select the *Association* tab of the settings notebook and add a file association by typing a wildcard in the *New name:* field and select *Add>>*. Delete the existing association in the same manner.

Windows: In Explorer, go to View, Options, and select the File types tab. Scroll down to the item associated with **.PAS** files and click Edit. Click **Open** in the actions window, and edit the filename of the executable to use to point to VP.EXE.

---

## Windows of the IDE

Several different types of windows can be found in the IDE. The most important one is the editor window, which is where programs should be written and where source level debugging takes place. A host of other windows are available, mainly to help find problems in your programs - a process called *debugging*. An introduction to the subject of debugging and the various facilities available in VP is available in chapter 6, starting on page 58.

### Local Menus

Most windows in the VP IDE have *Local menus* associated with them. To view the local menu for a particular window, either right-click the mouse in the window, or press Alt-F10.

Local menus typically contain shortcuts to much-used functions for the window; learning to use the local menus can save a lot of time.

Each menu item on a local menu has an associated hot-key (The letter is highlighted in the menu), making it easy to access a particular item once the menu has been displayed.

When you have used VP long enough that you start memorising the hotkeys, there is even a shortcut for reaching individual items on the local menu without first displaying the menu: Press Ctrl-Shift and the hotkey on the local menu. For example, the local menu for an editor window has an **O**rigin item, which causes the cursor to move to the current point of execution while debugging. Pressing Ctrl-Shift-O in an editor window has the same effect as pressing Alt-F10 to display the menu and then select the **O**rigin menu item.

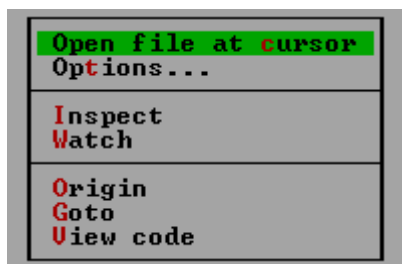


Figure 1: The local menu for an editor window

### The editor window: Colours

The editor window is where you write and edit your programs. In the default colour scheme, it contains a blue background with normal text being yellow. To change the colours to your liking, use the **O**ptions|**E**nvironment|**C**olours menu:



Figure 2: The colours dialog

In the editor window, a number of aids for writing Pascal programs are available:

### Cursor navigation

On the most basic level, you need to be able to navigate your source code as efficiently as possible. The VP editor implements the following keystrokes for cursor navigation:

Keys	Action
^S or Left arrow	Character left
^D or Right arrow	Character right
^A or Ctrl+Left arrow	Word left
^F or Ctrl+Right arrow	Word right
^E or Up arrow	Line up
^X or Down arrow	Line down
^W	Scroll up
^Z	Scroll down
^R or PgUp	Page up
^C or PgDn	Page down

### Editing

Even if it is nice to be able to move the cursor, it would not be much fun if the editor did not allow editing of text. Fortunately, it does. To add text, simply type it in the editor window and it will appear.

In addition, the following editing keys are available:

Keys	Action
^V or Ins	Insert mode on/off
^M or Enter	Insert line
^Y	Delete line
^Q H	Delete to start of line
^Q Y	Delete to end of line
^H or Backspace	Delete character left
^G or Del	Delete character
^Backspace	Delete word left
^T	Delete word right

## Block operations

Block operations are used to cut or copy manually selected fragments (blocks) of text to the clipboard, with an option to paste its contents back into the source file at a later time. The VPPM version of the IDE uses the OS/2 clipboard, so text copied to the clipboard can be pasted into other OS/2 programs and vice versa. Block operations can also be used for moving or indenting text in an easy and convenient manner, without interfering with the contents of the clipboard. (To indent is to move the text right or left).

VP gives you the choice of three different block types, although only one block type can be used at a time. The default block type is the stream block, which selects all characters between the start and the end of the block. Enable stream blocks by pressing **^KM**. The second block type is the line block, which selects text by the line instead of by the character. Press **^KX** to select this type of block. The third block type is the columnar block, enabled by **^KG**. A columnar block is a rectangular block of text.

A special clipboard window is used to hold the contents of the clipboard. Whenever you cut or copy text to the clipboard, the previous contents of the clipboard are flushed and the new text stored in its place.

The following keys apply to blocks:

Keys	Action	Keys	Action
<b>^KM</b>	Set stream block	<b>^KP</b>	Print block
<b>^KX</b>	Set line block	<b>^KI</b>	Indent block
<b>^KG</b>	Set columnar block	<b>^KU</b>	Un-indent block
<b>^KB</b>	Mark block begin	<b>^QB</b>	Move to beginning of block
<b>^KK</b>	Mark block end	<b>^QK</b>	Move to end of block
<b>^KT</b>	Mark single word	<b>^KD</b>	Exit to menu bar
<b>^KC</b>	Copy block	<b>^KL</b>	Mark line
<b>^KV</b>	Move block	<b>^Ins</b>	Copy to clipboard
<b>^KY</b>	Delete block	<b>Shift-Del</b>	Cut to clipboard
<b>^KR</b>	Read block from disk	<b>^Del</b>	Delete block
<b>^KW</b>	Write block to disk	<b>Shift-Ins</b>	Paste to clipboard
<b>^KH</b>	Hide block		

## Searching

For locating a particular string of text or for replacing one string with another, text search/replace is available. In addition to text search, the Search menu also provides options for finding the location of a specific procedure and for locating the last compiler or run-time error location.

Up to 10 bookmarks, or place marks, are also available. Setting a bookmark at a specific location in the source code makes it very easy to return to that location at a later time.

The following keystrokes are available for searching:

Keys	Action
<b>^Kn</b> (n = 0..9)	Set place marker (0-9)
<b>^Qn</b> (n = 0..9)	Go to place marker (0-9)
<b>^QF</b>	Find
<b>^QA</b>	Find and replace
<b>^QW</b>	Go to last error
<b>^L</b>	Repeat last find

### Undo and Redo

The VP editor allows you to undo or redo every editor action performed and maintains an undo buffer that can be as large as required. This gives you great freedom in experimenting with various features of the editor, without fear of losing lots of work, as you will be able to undo any changes made. Undo will reverse any operation done to your code, while redo is used to reverse any undo commands executed. The undo buffer is emptied when you leave the editor or close your source file, after which you will no longer be able to undo/redo.

Keys	Action
Alt-Backspace	Undo
Alt-Shift-Backspace	Redo

### Other editor options

In addition to the above mentioned editor features, the following keys are useful to know when programming in the IDE:

Keys	Action
^KS or F2	Save
F3	Open file
Alt-F3	Close active window
^I or Tab	Tab
^OI	Auto indent on/off
^Q	Pair matching (of brackets)
^OO	Insert compiler options in source file
^P	Ctrl+character prefix

### Macros

The IDE supports keystroke macros, where any key can be assigned a sequence of keystrokes. While recording a macro, the mouse can be used to perform actions that should not form part of the macro.

The scope of each macro can be global (i.e. it works in any context), or Editor only, in which case the macro only is available when an editor window is active.

Keyboard macros allow for detailed customisation of the user interface. For example, some users prefer a Turbo Debugger-style interface, and will redefine F2 to set a breakpoint, F9 to Run and Ctrl-F9 to Make. This is shown in the screenshot below, which shows the view displayed when selecting the Options!Macros!List dialog:



Figure 3: Three commonly defined keyboard macros

To define a new keyboard macro, use Options!Macros!Create (Alt=) to start recording. You are then prompted for the key to redefine; when this has been entered, a red **R** in the



upper right hand corner indicates that macro recording is taking place. Press the keys you wish to enter in the macro, and end recording by selecting Options|Macros|Stop recording (Alt-minus), preferably using the mouse. Pressing the key being defined again also ends macro recording.

If a mistake was made during recording, macros can be edited or deleted from the view shown in the figure above.

Note, that while recording a keyboard macro, all keys have their original meanings. Thus, it is possible to swap the meaning of the Ctrl-F9 and F9 keys as shown in the example above. However, this means that macros cannot be nested, and that it is not possible to define a macro for a key involving the key itself (as pressing it again stops recording).

This safeguards against circular or recursive macros that might cause the IDE to hang.

## Syntax Highlighting

Syntax highlighting makes it easier to get an overview of a piece of code. Reserved words are highlighted in white, comments are grey, numbers are green and strings are light blue. This makes it easy to spot spelling mistakes in keywords, since you expect them to be white. It also does away with the need for writing reserved words in UPPER CASE – an unnecessary habit of many programmers who started programming before Syntax Highlighting became a standard editor feature.

Syntax highlighting is enabled or disabled in the **Options|Environment|Editor** dialog in the IDE and is on by default, highlighting files with a .PAS or .INC extension.

Note, that syntax highlighting always assumes the source code is in **{&Comments+}** state. This means that nested { } comments are allowed and are syntax highlighted as such, up to 32 levels deep.

## Getting Help

To call for help on a menu item or other topics of the IDE, press F1 or select Help from the menu. To get context-sensitive help about the word currently at the cursor position, press ^F1. This will cause VP to search through the index and display help on the selected topic or the closest matching topic.

Online help for VP is provided in .VPH files. If you have installed Borland Pascal, you can make the BP help files available for use in the VP IDE by converting the Borland-style .TPH help files to the more compact .VPH format by using the Help Conversion utility. You can also write your own online help files and compile them for use with VP, using the Help Generator utility.

See the sections about The Help Generator HG and The Help converter TPH2VPH for more information.

On startup, VP looks for available help files in several places. Firstly, it looks in all paths listed in the VPHELP environment variable, followed by the ..\Common directory and the directory containing VP.EXE itself.

This way, help common to all platforms (such as IDE and RTL help) does not have to be duplicated, and platform-specific help (such as API references) can be copied to the relevant \bin.%p directory.

## The menu bar

The menu bar appears on the topmost line of the screen and provides access to most features of VP. It has been designed to look familiar to people used to working with

computer software and divides the available commands into logical sections of related commands.

To activate the menu bar at any time, either press **F10** or click on it using the mouse. A combination of pull-down and pop-up menus then gives easy access to all menu commands and options.

In the following, the commands of the above mentioned logical groups are outlined. Because the size of this manual is limited, some basic knowledge of files and programs is assumed along with a basic understanding of the operating systems.

The **File menu (Alt-F)** offers choices for opening and loading existing files, creating new files, changing directory, as well as saving files. It also provides the ability to shell to a command prompt window (temporarily leaving the IDE), print the active source file and exit the IDE.

The **Edit menu (Alt-E)** provides commands to cut, copy and paste text in edit windows. You can also open a clipboard window to view or edit its contents.

The **Search menu (Alt-S)** provides commands to search for text, procedure declarations and error locations in your files.

The **Run menu (Alt-R)** provides commands to run a program and to start and end debugging sessions. To use any of the Run commands except **Run|Parameters**, you must have compiled and linked your program. For details on using the debugging features of the IDE, please refer to chapter 7.

The options available from the **Compile menu (Alt-C)** are used to turn source code into an executable program or a dynamic link library. It is possible to reduce the amount of time required for compiling and linking by selecting the method most suitable for your needs:

- **Compile|Compile** compiles only the currently active file, even if several source files are loaded.
- **Compile|Make** compiles the primary file (if specified using **Compile|Primary File**) or the file in the currently active editor window, as well as all changed files on which the primary file depends. This option is the one normally used.
- **Compile|Build** compiles all files in a project, regardless of whether any changes have been made to the source files.

Additionally, the **Compile menu (Alt-C)** is used to select the desired target platform, OS/2 or Win32. Every time the menu item is selected, the value toggles to the next available platform.

The **View menu (Alt-V)** allows easy access to several powerful views, each containing information about different aspects of a program. The items in this menu are useful when debugging a program; for information about the options available, please refer to the chapter on debugging.

The **Debug menu (Alt-D)** is also used when debugging programs. Again, please refer to chapter 7 for more information on this subject.

The **Options menu (Alt-O)** contains commands that let you view and change various default settings in the IDE. Most of the commands in this menu lead to a dialog box. Please refer to the section 'Configuring the IDE' for an explanation of the most commonly used options, or refer to the online help for details.

The **Window menu (Alt-W)** contains commands for manipulating, arranging and opening editor windows.
















The **Help Menu (Alt-H)** can be used to browse the online help. **Help|About** displays a dialog box detailing the current version of VP being used.

## The VP/PM Smartbar

The OS/2 VP/PM IDE has a list of icons at the top of the window. Each of these icons correspond to a menu item in the IDE. Clicking one of the icons has the same effect as choosing the menu item.

To speed up memorising the icons, you can rest the mouse pointer “on” one of the icons. After about one second, a small yellow box with a short description of the button will appear.

The icons available are the following:

Icon	Command	Keyboard Shortcut
	Open File	F3
	Save File	F2
	Undo last editor operation	Alt-BackSpace
	Cut selected text to clipboard	Shift-Del
	Copy selected text to clipboard	^Ins
	Paste text on clipboard	Shift-Ins
	Search for text in editor window	^QF
	Repeat last search	^L
	Compile current source file	Alt-F9
	Make current unit	F9
	Run program	^F9
	Trace into current call	F7
	Execute and step over current call	F8
	Stop current action (Compilation or execution)	^Break
	Get help	F1

## Configuring the IDE

The IDE is highly configurable, and can be customised in a number of ways. Using the **Options** menu, a number of options for configuring all aspects of the IDE and the compiler are available.

A description of the configuration options in the **Options|Compiler** and **Options|Linker** dialogs can be found in the following chapter. These dialogs are used for configuring the behaviour of the compiler, and are not relevant to this chapter.

### Options|Resource compiler

Most graphical applications use resources in one form or another. Resources are anything from menus to buttons, dialog definitions and icons, to tables of strings and help identifiers.

The resource information for GUI applications is typically stored in *resource files*, which are just plain text files of a special format that describe the resources of the applications. Before the resources described can be linked into the program, the resource files need to be compiled to binary form - this is done using a resource compiler. The job of the resource compiler is to turn the resource file (which should have a .RC extension) into a binary .RES file and link the .RES file into your executable.

VP can use any resource compiler available; in OS/2, the Resource Compiler (RC.EXE) that comes with OS/2 is used. If you have moved it to another location, or wish to use another resource compiler, the entry field in the **Options|Resource compiler** dialog should contain the full path specification of the resource compiler.

Creating resource files can be done either by hand, or by using one of several readily available tools for this purpose.

Resource files are included in a program by using the **{\$R ResFile.Res}** compiler directive. See also the **{&Open32}** directive.

For Win32, an executable named RC.EXE is included in the bin.w32 directory. This executable accepts parameters compatible with OS/2's RC.EXE, and passes control to Borland's BRC32.EXE resource compiler (with appropriately translated parameters). For this to work, BRC32.EXE and files required by this executable must be available in PATH.

### Options|Directories

This dialog contains a list of directory-names; each of the directories listed must already exist. All of the entry fields (except the two topmost ones, containing output directories) can contain a list of directories, separated by semicolons (;).

The **Output directory** is where the VP compiler stores all supporting output files, typically files with OBJ, LIB, VPI, LNK and RES extensions. This directory is also searched if a unit is not found in the list of **unit directories**.

The **Exe output directory** is where the compiler creates the real output (.EXE and .DLL files) along with their .MAP files, if map file creation is enabled in the linker options.

The **Include directories** contains the list of directories where the compiler searches for *include files* included using the **{\$I FileName}** compiler directive. Before any of the directories listed are searched, a matching file in the current directory will be included.

The **Unit directories** entry field contains the list of directories the compiler searches for *unit files* included in a program or library through a **uses** clause. The current directory is implicitly added to the beginning of the list and is searched before any other directories. Unit directories can contain either Pascal source code or precompiled units in the form of .VPI files with associated .OBJ or .LIB files.

The **Library directories** is where the linker searches for .LIB files when resolving references to identifiers declared as **external**. Object files (.OBJ) included with a **{\$L ObjectFile}** should also be in directories listed here. As for Include and Unit directories, the current directory is searched first.

The **Resource directories** are where the Resource compiler is instructed to look for .RC, .H and .DLG files that are included into the main resource file, as well as precompiled .RES files.

For environments where these directories are shared between different operating systems, Virtual Pascal implements support for “Drive Substitutions” to make it easier to manage the process. For more information on this, please refer to the chapter about [Extending Virtual Pascal](#).

## Options | Environment | Preferences

This dialog is used to set up certain aspects of the editor and the IDE in general.

The **Screen size** option applies only to the text-mode version of the IDE (VP.EXE) and is used to control the window or screen size of the IDE at start-up. You can select either 25 or 43/50 lines, corresponding to DOS text mode windows, or you can select **Keep original**. In this case, VP will keep the window size set by a previous **mode** command on the command line. In this way, you can execute **mode co100,40** (OS/2), **mode 100,40** (Windows) and then run VP in a 100x40 window.

The OS/2 VP/PM IDE always uses **Keep original**, and may be resized at will, up to a maximum window size of 150x50 characters.

**Source tracking** determines whether a new edit window should be used to display source code as you step through a program during a debugging session. If the source file referenced is not already open in an edit window, the IDE opens it and highlights the appropriate source line. If **New Window** is selected, a new editor window will be opened, and if **Current Window** is selected, the newly loaded source file will replace the contents of the current edit window.

**Auto save** is used to make the IDE save some or all of the current desktop configuration automatically. If **Editor files** is checked, the currently active files will be auto-saved on every make, build, compile or shell command. The **Environment** checkbox covers all settings contained in the Options menu, any changes made to the search dialog boxes and the primary file name entry. If the **Desktop** option is selected, the position, size and cursor position of all currently open windows will be stored when the VP IDE is exited.

Do note, that editor files are saved independently of the current **Editor files** setting when compiling, making or building a program. This is required because the compiler is an external program (VPC.EXE) that always uses the source files currently on disk for compilation.

If **Change dir on open** is checked, the IDE changes the current directory when a new file is opened.

If **Return to last dir** is checked, the IDE stores the current directory and changes to this directory when VP is next started.

If **Bring messages to top** is checked, the Messages window (which displays progress information for the compiler, linker, etc) will be shown and made active when a compilation is started. The messages window can still be closed by pressing ESC or ALT-F4. If this option is not checked, the Messages window behaves like any other window.

Note, that the Messages window must be the active window for the IDE to be able to position the cursor at the position of any errors occurring during compilation.

## Options | Environment | Editor

Using this dialog, the editor can be extensively configured. Each of the configuration options available will be covered briefly; for a more exhaustive explanation, please refer to the online help.

If **Create backup files (default ON)** is checked, a backup copy is automatically created when a file is saved. This file is given a .BAK extension, independent of the extension of the original file.

**Insert mode (default ON)** determines whether the editor is in Insert or Overtyping mode when it is first started. The insert mode can be changed while editing by pressing the Ins key.

In **Auto indent mode (default ON)**, pressing Enter causes the editor to move to the next line, in the column containing the first non-blank character in the line above. This is very useful when writing Pascal programs, where the indentation usually changes only slightly from one source line to another. If this option is unchecked, pressing Enter places the cursor in column 1 of the next line.

**Smart tab (default ON)** is another valuable help when writing Pascal programs. When this option is checked, TAB moves the cursor to the next non-space character in the line above the current one. If it is unchecked, TAB just moves the cursor to the next tab position as defined in the **Tab size** input box.

If **UNIX text (default OFF)** is checked, the editor handles UNIX style text files correctly. In normal MS-DOS text files, each line is terminated by Carriage Return (#13) followed by LineFeed (#10), whereas UNIX text files use only the LineFeed (#10) character to separate lines.

When **Optimal fill (default OFF)** is enabled, all long sequences of spaces are converted to TABs to make the source file as small as possible. In the disabled state, all TABs are converted to spaces before the file is saved.

When **Backspace unindents (default ON)** is checked, pressing the Backspace key when the cursor is on a blank line or on the position of the first non-blank character of the line, the line is outdented to the previous indentation level. If this option is unchecked, Backspace simply deletes one character.

When **Persistent blocks (default ON)** is selected, marked blocks remain marked if a key is pressed. When this option is off, pressing any key de-selects the block.

**Truncate spaces (default ON)** causes the IDE to remove spaces that trail lines of text, immediately after loading or before saving a file. Spaces are also truncated when the End key is pressed.

**Syntax highlighting (default ON)** colours the various elements of Pascal source code differently. This significantly improves the readability of code and helps to avoid syntax and spelling errors. When this option is off, all code appears with the default text foreground and background attributes. The colours used for Syntax highlighting can be configured in the **Options|Environment|Colours** dialog, described on page 39. Also refer to the **Highlight extensions** setting on this page.

Enable **Block insert cursor (default OFF)** to change the way the cursor looks. In the default OFF state, a block cursor indicates Overtyping mode, and an underline cursor indicates Insert mode. Checking this option reverses this.

If **Find text at cursor (default ON)** is checked, the word underneath the cursor appears as the text to search for in the **Text to find** input box of the **Search|Find** and **Search|Replace** dialogs. If OFF, the search text of the previous search will appear.

**Background only (default OFF)** determines how a selected block is displayed. In the OFF state, all text in a block appears with the same fore- and background colours. In the ON state, only the background colour changes, and syntax highlighting, breakpoints etc are still visible.

When the **Overwrite Block** option is checked, the IDE behaves closer to CUA standard. When enabled, this option causes a double mouse click to select the current word instead of the current line.

In addition, when **Persistent Blocks** are disabled (and this option is enabled), and the cursor is inside or at the beginning or end of the selected block,

- Pressing the DEL or Backspace keys causes the block to be deleted,
- Typing any character key causes the block to be replaced with that key,
- Pasting from the clipboard replaces the selection.

The **Highlight extensions** entry field contains a list of file wildcards, separated by semicolons (;). If Syntax highlighting is enabled, all files matching a wildcard from this list will be syntax highlighted.

### Options | Environment | Mouse

This dialog is used to configure the default behaviour of the right mouse button in combination with the Ctrl, Alt and Shift keys. Each combination can be assigned to an action described elsewhere in this manual.

If the **Reverse mouse buttons** option is enabled, the meaning of the right and left mouse buttons are reversed.

Setting the **Mouse double click** slider can set the speed at which two consecutive mouse clicks are considered to be a single double-click instead of two distinct single-clicks.

The default settings when holding down a key while clicking the right mouse button are:

Key	Action	Keyboard shortcut
Ctrl	Context-sensitive help index lookup	^F1
Alt	Execute to Cursor	F4
Shift	Inspect	Debug Inspect or Ctrl-Shift-I

### Options | Environment | Colours

Using this dialog, all colours used in the VP IDE can be changed to your liking. The colours are divided into several groups: *Desktop*, *Menus*, *Dialogs*, *Editor*, *Help*, *Data window*, *Low level*, *Breakpoints* and *Syntax*.

Each group contains a number of items, the colour of which can be modified separately. To change a colour setting, click on the item you wish to change, and click on the desired foreground and background colours. A sample of the setting can be seen in the lower right-hand corner of the dialog.

When all colours are to your liking, select OK. Selecting Cancel causes all changes made to be lost.

### Options | Save options

This dialog allows you to save the current desktop layout and environment settings in a VP options file with a default extension of .VPO. This file can be retrieved later by specifying the file name on the command line as described on page 27, or it can be loaded using the **Options|Retrieve Options** menu item.

This effectively implements complete project management. You can store as many different configuration files as you want, and have different settings for directories, colours, compiler, etc for each of them.

## **40 The VP User Interface**

---

By being able to load and save different IDE settings and desktop layouts, the task of working on several different projects at the same time is greatly simplified.

The .VPO file is a text file, editable with a normal text editor like the VP IDE. The .VPO file is not compatible with the binary .CFG file format used by earlier versions of Virtual Pascal, although VP v2 is still able to read these, just not write them. When loading an old-style .CFG file, make sure it is given a .VPO extension when it is re-saved.



## Writing programs using VP

This chapter guides you through the steps required to create simple programs in VP, and discusses the different compiler options available. You should be familiar with the principles of starting, using and configuring the IDE introduced in the preceding chapter before reading this.

This is not meant as a tutorial in Pascal programming. If you are unfamiliar with the Pascal language, please refer to the Language Reference section of this manual, or third-party literature. Several books covering Pascal, Pascal programming or OS/2 and Windows specific programming issues exist; this chapter is meant solely as an appetising introduction to the topics in question.

If you are an experienced programmer with knowledge of Borland Pascal, you may want to skip the first section of this chapter and go directly to the section discussing configuring the compiler on page 42.

### Writing your first program

Start the VP IDE by clicking on the appropriate icon in the VP desktop folder or the Start Menu.

Select **File|New** to open a new editor window, and type in the following in the editor window:

```
Program Hello;  
{&PMTType:VIO}  
begin  
  WriteLn('Hello World!');  
  ReadLn;  
end.
```

This is a complete Pascal program!

The first line defines the program name as being *Hello*.

The second line instructs the linker to create a text-mode (VIO) application that can be run either in a window or in full-screen. The same effect can be achieved by selecting **Compatible with PM** in the **Options|Linker|Applications** dialog box.

The program itself starts on line 3, and consists of just two statements enclosed by **begin** and **end**. The *WriteLn* statement is a built-in function defined in the *System* unit of VP, and causes the string passed to it as a parameter to be output on the screen. The *ReadLn* statement causes the program to wait for the user to press Enter.

Let us save the new program by pressing F2. The File Save dialog pops up, and asks for a name for the file. Type in `Hello` to save the file as `Hello.Pas`.

The next step is to have VP create an executable file (.EXE) that can perform the actions instructed. This is achieved in a single step, since VP handles the details of compiling and linking in a transparent manner. First, make sure the correct target platform is selected by checking the **Compile|Target platform** option. If it is not, just select it to toggle between the available platforms.

Press F9 or select the **Compile|Make** menu item to start the compile/link process.

## 42 Writing programs using VP

---

This will cause the *Messages* window to be opened at the bottom of the screen (if the **Bring messages to top** option is enabled. This window keeps you informed of the progress of the compilation process, and should contain something like this:

```
6 lines, 0.1 seconds.  
Linking...  
Success.
```

You have now created your first executable program! In order to execute it, either open a command line window, change the current directory to be identical to the **Exe output** directory, and type `Hello` to execute the newly created `HELLO.EXE`. This program is a stand-alone executable that you can copy to a disk and execute on any computer running your target operating system - no unwieldy DLLs need to be copied along with it.

Alternatively, run the program from within the Virtual Pascal IDE. To do so, press `^F9` or select the **Run|Run** menu item.

## Your second Virtual Pascal program

Let us expand the program slightly: Change the program to read

```
program Hello;  
{&PMTType VIO}  
uses  
  Crt;  
begin  
  ClrScr;  
  GotoXY( 10, 10 );  
  WriteLn('Hello World!');  
  ReadLn;  
end.
```

This version of the program introduces a new concept: **units**. A unit is a Pascal source file, which uses the reserved word **Unit** in place of **Program**. Other Pascal programs or units can *use* the functionality of the unit

The **uses** clause on line 3 tells the compiler that the program requires part of the functionality defined in the *Crt* unit. The *Crt* unit forms part of the standard Run-Time Library that comes with VP, and implements a large number of functions dealing with screen in- and output.

The two new statements in the program, *ClrScr* and *GotoXY*, are both defined in the *Crt* unit. *ClrScr* causes the screen to be cleared, and *GotoXY* places the cursor at the screen location specified as parameters.

Now press `^F9` again. VP realises that the source code has changed, and re-compiles and re-links the program before executing it. This time, the program actually clears the screen before it writes the text - and the text is located at position (10,10) on the screen.

## For more information

A detailed description of the structure and formal syntax of a Pascal program, along with a description of all compiler directives, reserved words, etc, can be found in the Virtual Pascal Language Reference Manual.

Note, that the Language Reference is not a Pascal textbook; for a proper introduction to writing Pascal programs, it is recommended that a book on this subject is purchased separately.

## Configuration and tools

The VP compiler can be configured for a number of different purposes. While you are in the process of writing a program, you will want as much information to be available as possible, and do not care that much about the size of the program.

When you have finished writing your program, you will probably want all extraneous information to be removed and the program to be as small and execute as fast as possible.

This chapter provides information about the use of the various settings available, and also covers the command-line tools included with Virtual Pascal.

### Changing settings in the IDE

The Virtual Pascal compiler is extremely configurable, and can be set up in a number of different ways, depending on what you are currently doing. This process is performed via the **Options|Compiler** and **Options|Linker** dialogs.

#### Options | Compiler

This is a huge dialog, where many aspects of the compilation process can be configured. To ease setting up the most commonly used configurations, three default settings are provided: *Fastest*, *Smallest* and *Debug*.

Select the option you require by pressing the appropriate button; the buttons are located in the lower left hand corner of the dialog.

##### Fastest setup

Choose this if you have finished debugging your application and want the compiler to generate the fastest possible executable, even at the expense of file size.

##### Smallest setup

Press this button when you have finished debugging your application and want the compiler to generate the smallest possible executable, at the expense of speed.

The difference between Fastest and Smallest setup is usually relatively small. It is recommended that you experiment with these settings to see which one best suits your particular application.

##### Debug setup

Select this, if you are still in the process of writing and debugging your application. The program will be both slower and larger than for the *Fast* and *Small* setups, but a host of information enabling you to find problems in your code will be available. For details on how to debug applications in VP, please refer to the following chapter.

#### The individual configuration options

Once you have selected your preferred overall setup using one of the three defaults, you can fine-tune the setup by manipulating the settings individually. Also, some of the settings available do not influence code size, speed or debug information, and are not changed by the default selection buttons.

The options are divided into several related groups, and each option is covered in the following.

## 44 Configuration and tools

---

The **Code Generation** group of checkboxes tells the compiler how to compile your code. Each option is equivalent to a switch compiler directive, with the checked state of the check box corresponding to the enabled (+) state of the directive. All VP compiler directives are explained in detail in the Language Reference section of this manual and in the online help. For each option, the corresponding compiler directive is listed along with its default setting. The following options are available:

- Align Code, **{&AlignCode+}**. When this is enabled, **procedure** and **function** entry points are aligned on 4-byte addresses. This slightly speeds up execution, at the cost of code size.
- Align Data, **{&AlignData+}**. In the enabled state, variables and typed constants are aligned based on their size. *Byte*-sized variables are not aligned, *Word*-sized variables are aligned on 2-byte boundaries, and variables with a size greater than 2 are aligned on double-word (4 byte) boundaries. In the disabled state, no data alignment is performed. Aligning data in this way can speed up execution speed.
- Align rec/obj, **{&AlignRec-}**. When enabled, individual fields in record and object types are aligned according to the same rules as for **{&AlignData}**. In the default (disabled) state, no field alignment takes place. Exercise caution with this setting. If a record is defined as:

```
TestRec = record
  b : Byte;
  w : SmallWord;
end;
```

the size of the record will be 3 in the **{&AlignRec-}** state, and 4 in the **{&AlignRec+}** state, since the *SmallWord* size variable will be aligned to offset 2 within the record; the byte at offset 1 will be unused.

In the **{&AlignRec+}** state, alignment can be disabled for a record declaration by using the reserved word **packed** in the type declaration. To make sure the size of the record is as small as possible, the following definition can be used:

```
TestRec = packed record
  b : Byte;
  w : SmallWord;
end;
```

In this example, the size of the record is 3 bytes, irrespective of the setting of **{&AlignRec}**.

- Asm stack frame, **{&Frame+}**. This setting specifies whether or not to create a stack frame for **assembler** procedures. Please refer to the Language Reference section for a detailed description of the implications of this setting.
- Original names, **{&OrgName-}**. It should only be required to change this setting when creating VP object files or DLLs for use in programs written in other languages. In the disabled state, the unit name and a @ character are prepended the identifier names in the generated object modules. This allows for identifiers with the same name, as long as they reside in different units. In the **{&OrgName+}** state, the names of identifiers are converted to upper case and are not prepended with the unit name.

See also the chapter on Naming Conventions in the Language Reference Manual.

- Stack Frame, **{SW+}**. Controls the generation of stack frames for normal Pascal procedures and functions. In the default state, a stack frame is always generated. In the disabled state (automatically selected with the *Smallest* option), stack frames are only generated when they are required. If a **procedure** or **function** does not have a stack

frame, certain debugging options are not available (Call stack display and local variable evaluation).

- DWord enums **{Z-}**. In the default state, enumerated types take up only one byte of storage, if the number of elements is smaller than 256, and 2 bytes for sets containing 256 elements or more. In the enabled state, enumerated types always occupy 4 bytes.

The **Calling convention** radio buttons let you specify the default conventions used to pass arguments when calling functions and procedures:

- Pascal: This is the default, and should normally be selected. An individual procedure can be made to use this calling convention by appending the **pascal**; standard directive after its definition.
- C, **{CDecl-}**. Enables the C style calling convention, and should be used for procedures that are called by OS/2, such as dialog handlers in PM programs. An individual procedure can be made to use this calling convention by appending the **cdecl**; standard directive after its definition.
- 16-bit Pascal, **{Far16l-}**. Enables the far 16-bit Pascal calling convention, which should not normally be used. An individual procedure can be made to use this calling convention by appending the **far16**; standard directive after its definition.
- Standard, **{StdCall-}**. Enables the standard calling convention, and should be used for procedures called by Windows, such as dialog and message handlers. An individual procedure can be made to use this calling convention by appending the **stdcall**; standard directive after its definition.

The **CPU** radio buttons specify which instructions are allowed in **asm..end** statements. The following options are available:

- 80386 only, **{G3+}**. In this state, only instructions available on 80386 and later are allowed. This is the default.
- i486, **{G4+}**. In this state, instructions available on either 80386 or i486 processors are allowed. The compiler may generate code that runs faster or is smaller on i486 processors (but will still run on 386 CPUs).
- Pentium, **{G5+}**. In this state, all instructions available on 80386, i486 or Pentium processors are allowed. The compiler may generate code that runs faster or is smaller on Pentium processors (but will still run on 386 and 486 CPUs).

The **Optimize for** buttons determines the compiler's code generation strategy:

- Optimise for Size, **{Size+}**. In the enabled state, the compiler always chooses the smallest code sequence possible at the expense of execution speed.
- Optimise for Speed, **{Size-}**. In this state, the compiler optimises for speed, and chooses the fastest sequence possible with less regard for code size.

The **Optimisation** group of checkboxes tells the compiler whether to optimise the generated code:

- Optimise, **{Optimise-}** or **{Optimize-}**. In the disabled state, the compiler performs no code optimisations. In the enabled state, the compiler uses a number of strategies to maximise execution speed *and* minimise code size. This is done by a sophisticated combination of optimisation methods, including routines for removing redundant instructions and storing local variables in registers.

*Note:* For very complex programs, it may be necessary to disable optimisation while debugging the program. If a local variable is stored in a register and the compiler

## 46 Configuration and tools

---

runs out of registers to allocate, the register variable may be temporarily *spilled* to memory. The integrated debugger does not have information that this has happened, and will (wrongly) display the register value when watching, evaluating or inspecting the spilled variable. This can cause strange results when manually changing the value of a spilled variable.

- Smart link, **{&SmartLink-}**. In the disabled state, all code and data from all units **used** in the program or library are linked into the executable. In the enabled state, VP compiles each unit in a way that allows the linker to only include code and data that is actually being used. This typically reduces the size of the executable by a very significant amount. There is a small overhead in linking speed when smart linking is enabled.

Note, that smart linking actually takes place already at compile time (the compiler generates **.LIB** files when smart linking is enabled and **.OBJ** files when it is disabled), so it is not enough to simply turn Smart linking on and *Make* the project to switch from one setting to another. It is possible to mix the two, with some units being **.LIB** files and some **.OBJ** files.

The **Runtime Errors** group of checkboxes specifies the type of error checking code the compiler generates. Error checking slows down execution speed and results in slightly larger executables.

- Range checking, **{SR-}**. When this option is enabled, all indexing of string or array types is checked for out-of-bounds conditions, as are all assignment operations to scalar and subrange types. If a check fails, a run-time error 201 is generated.
- Stack checking, **{SS-}**. In the enabled state, all procedures and functions check to see if there is enough room on the stack to complete the call. If there is not, a run-time error 202 is generated. Do not disable Stack checking, unless you are absolutely certain that a stack error can not occur. Stack overflow errors in the **{SS-}** state can be very hard to find.
- I/O checking, **{SI+}**. In the enabled state, any I/O operation (such as reading from a file) that fails will cause a run-time error to occur. In the disabled state, the result of I/O operations can and should be accessed by checking the *IOResult* function.
- Overflow checking, **{SQ-}**. In the enabled state, certain integer functions such as +, -, \*, Abs, Pred, Succ and Sqr are checked to prevent the result from overflowing the allowed range. If an overflow occurs, a run-time error 215 is generated. In the disabled state, no overflow checking takes place.

The **Syntax Options** group of checkboxes specifies the type of syntax options you want the compiler to use and recognise.

- Strict **var** strings, **{SV-}**. In the enabled state, the size of the formal and the actual parameters must be identical when passing strings as variable parameters. In the disabled state, it is the responsibility of the programmer to ensure that no critical data is overwritten, even if the declared length of the string passed is different from the declared length at the receiving end.
- Complete boolean evaluation, **{B-}**. In boolean expressions, the result of the entire expression is often evident before the expression has been fully evaluated. In the default **{B-}** state, evaluation of the expression stops as soon as the value of it has been determined. In the **{B+}** state, every sub-expression is guaranteed to be evaluated.

In the **{B-}** state, code that would otherwise be illegal can be executed successfully. For example,

```
while ( p <> nil ) and ( p^ = 8 ) do
  ...
```

would result in a run-time error in the **{SB+}** state, since evaluating  $p^$  when  $p = \text{nil}$  is an error.

However, if the evaluation of an expression involves calling a function that has a side-effect (i.e. affects the state of the program), **{SB+}** may be appropriate. For example,

```
if ( x = 5 ) and ( IOResult = 0 ) then
  ...
```

should be evaluated in the **{SB+}** state, since *IOResult* is a function that returns the value of the **threadvar** *InOutRes* and sets it to 0 – thus changing the state of the program.

It is generally not advisable to write code that relies on a certain state of the **{SB}** directive.

- Extended syntax, **{SX+}**, enables a set of extensions to the Pascal language that were introduced in Borland Pascal v7. These include:
  - Functions can be called as procedures; in this case, the function result value is simply discarded.
  - Handling of null-terminated strings is enabled, and 0-based **array of char** types can be treated as 0-terminated strings.
  - If in the **{&Delphi+}** state as well as the **{SX+}** state, function results can be assigned to the implicitly defined *Result* variable. *Result* has the same type as the function result, can be used as any other variable, and can even be passed as a variable parameter to another function or procedure.
- Typed @ operator, **{ST-}**. When this option is disabled, the @ operator returns an untyped **pointer** that is compatible with all other pointer types.

In the enabled state, applying @ to a variable of type T returns a value of type  $T^$ , and normal type checking rules apply.

- Open string Parameters, **{SP-}**. In the enabled state, all **string** type parameters are considered to be open strings. This means that the routine called can access information about the declared length of the string actually passed. In the default disabled state, **string** parameters are interpreted as normally.

In any case, the *OpenString* identifier can always be used to explicitly make a parameter an open string type.

- Delphi syntax, **{&Delphi+}**. In the default enabled state, the language extensions introduced by Borland Delphi v1, and some features from v2 and v3, are allowed in VP programs. For a list of these features, please refer to page 16.

Since these extensions include adding new reserved words, old code not written for these extensions may not compile in the **{&Delphi+}** state. To disable any of the new features that could break old code, you can temporarily switch to the **{&Delphi-}** state.

The new reserved words are **as**, **class**, **except**, **finally**, **finalization**, **initialization**, **is**, **on**, **property**, **raise**, **threadvar**, and **try**.

- Huge strings, **{SH-}**. This option controls the meaning of the reserved word **string**. In the enabled state, **string** is equivalent to *AnsiString*, a dynamically allocated string type with a length of up to 2GB. In the disabled state, **string** is equivalent to

## 48 Configuration and tools

---

*ShortString*, the standard static Pascal string with a maximum length of 255 characters.

- Writeable **consts**, **{&J-}**. Controls write access to typed constants. In the enabled state, typed constants are treated as true constants that cannot be modified, parsed as **var** parameters, etc. In the default (disabled) state, typed constants are treated as initialised variables.
- Open32 compatibility mode, **{&Open32-}**. This switch, which only applies when the target platform is set to OS/2, is intended to assist the process of porting Windows applications to OS/2. In the enabled state, resource files included using the **{&R ResFile}** directive are treated as Windows (Win16 or Win32) resources, and are automatically converted to OS/2 format prior to the link process. Additionally, the **stdcall** directive is changed to have the same meaning as the **cdecl** standard directive. To declare a function as using the **stdcall** calling convention in the **{&Open32+}** state, use the **{&StdCall+}** directive.
- Automatic inclusion of the *Use32* unit, **{&Use32-}**. This directive can be used when porting 16-bit Borland Pascal applications to Win32 and OS/2. In the enabled state, the *Use32* unit, which redefines the meaning of the basic *Integer* and *Word* types to be 32 bits, is implicitly included in the interface section **uses** clause of every unit. In this state, the *Use32* unit must not be explicitly mentioned in the **uses** clause.
- Nested “curly brackets” comments, **{&Comments-}**. When this directive is enabled, the compiler allows **{ }** comments to be nested, up to 32 levels. Because this switch may break code containing mismatching comments, the default state is Off.

The **Debugging** group of checkboxes turn on or off the generation of information used for debugging.

- Debug information, **{&D+}**. When this option is enabled, the compiler collects information about the correlation between line numbers and addresses in the executable module. When this information is available, source level debugging (as described in more detail in the next chapter) is possible. This directive also encompasses the old **\$L** directive, which is ignored in VP v2. When **\$D** is enabled, local symbol information is collected and stored with the line number information, meaning that variables can be evaluated and inspected when debugging an application. Please refer to the next chapter for more information about debugging.
- Line numbers, **{&Zd-}**. In the enabled state, information about line numbers is stored in the executable file in a standard format that can be used by other debuggers, such as IBM's IPMD. VP's integrated debugger does not use the information collected in this way.
- Assembly output, **{&Asm-}**. Enabling this option only works if smart linking is disabled: **{&SmartLink-}**. In the enabled state, an assembler file (.ASM) is created for each unit compiled in this state. This file contains the assembler mnemonics that VP has generated from the Pascal source code, and can be compiled by standard assemblers such as MASM and TASM to produce object files. In addition, you can use the assembler files to look at the code generated by VP, and maybe find ways to optimise your code.

In the **Conditional Defines** input box, you can enter a list of conditional defines, separated by semicolons (;). These symbols are used in conditional compilations, where code should only be compiled if a particular symbol is defined (or not). For more information on conditional compilation, please refer to the Language Reference section. As an example, the entry field could contain

*BetaVersion;DebugCode*



This defines two symbols, `BetaVersion` and `DebugCode` for use in conditional compilation with `{$IFDEF BetaVersion}` or `{$IFDEF DebugCode}` defines.

The **Unit aliases** entry field can contain a list of unit names and the unit for which each should be substituted. This is useful when porting Win16 applications to Win32, since these typically include the `WinTypes`, `WinProcs` and `WinApi` units in the **uses** clause and these units do not exist in Win32. Instead, a set of 3 unit aliases can be entered, aliasing each of these units to the `Windows` unit, including the definitions of them all, like this:

```
WinTypes=Windows; WinProcs=Windows; WinApi=Windows
```

The **Stack size** entry field contains the number of bytes to allocate for the stack. Enter a value that is no less than 8192 and no greater than 512MB. Typically, the default value of 16kB is sufficient. This setting is overridden by a **\$M** directive in the source code (See the Language Reference for details of this and other compiler directives).

## Options | Linker

VP includes a powerful one-pass built-in linker, and also has the ability to use an external linker. Note, that some debugging options such as Run-time Location Information are only available when using the built-in linker.

The job of the linker is to combine the object files created during compilation (.OBJ and .LIB files) to an executable image for the target operating system. VPLink is capable of generating both OS/2 LX and Win32 PE executables (both .EXE and .DLL files).

The **Application** radio group specifies the type of application being generated.

- When compiling for OS/2, **Full Screen** generates text-mode applications useable in screen sessions only. In Windows, this setting is identical to the **Compatible with GUI** option. This option is equivalent to using the `{&PMTType:NoVio}` compiler directive in the program file.
- **Compatible with GUI** should be used when creating a text-mode application that can run either in full screen or in a window on the desktop. Checking this box corresponds to using the `{&PMTType Vio}` compiler directive.
- **GUI application** instructs the linker to create a graphical application, native to the target operating system, such as those created when using OWL or Borland Delphi.. Checking this box corresponds to using the `{&PMTType PM}` compiler directive.

The **Compression** group determines whether to compress the executable and any resources in it. This setting applies to (OS/2 only, as Windows does not support compressed executables; for a Windows target, this setting is ignored).

- Select **None** to not perform any compression.
- Select **Exepack 1** to compress the executable using minimum compression, supported by all versions of OS/2 from v2.0 and up.
- Select **Exepack 2** to compress the executable using a superior compression algorithm. This compression method is only supported on OS/2 Warp and later; programs linked with this compression will not run on OS/2 v2.x.
- When the **ExePack 2** option is selected and the target is Win32, the executable is marked as “Windows v4.0” and will not run in older Win32 variants, such as Windows NT v3.51. When **ExePack 2** is not selected, the Win32 executables will be marked as “Windows v3.10” and will run in any version of Win32.

## 50 Configuration and tools

---

The **Map File** radio buttons specify which type of map file should be generated for the executable. A map file is a plain text file containing information about the executable, segments used, line numbers, etc. When enabled, the map file is stored in the **Exe output** directory along with the executable itself.

- To disable creation of a map file, select **None**.
- To generate a map file that only lists the segments of the executable, select **Segments only**.
- To generate a map file that includes both segment information as well as a list of all public symbols in the executable, select **Public**.
- To generate a full map file, containing the same as above, plus the addresses of entry points for all functions and procedures, information about imported and exported functions and detailed line number information, select **Detailed**.

The **Options** checkboxes determines other linker settings:

- To include Location Information in the executable, check the **Generate location info** option. When this option is enabled, the linker will store location information for every unit compiled in the **{&LocInfo+}** state. This information consists of the name of the source file and line number information, which can be used for locating errors in the program or for other purposes, using the *GetLocationInfo* function of the *System* unit.

The overhead of including location information is about 1 byte per source line. Location information is only available when using the built-in linker.

- To use an external linker instead of the built-in one, check the **Use external linker** option. and configure the linker in the entry fields below.

When the external linker is OS/2's LINK386, the current linker settings are reflected as command line options in the **LINK386 options** entry field. If a linker other than LINK386 is used, all desired command line options should be entered in the **other options** entry field. When using an external linker, the name of the linker executable should be entered in the **Linker path** entry field.

### Generating an import library

An import library is a .LIB file that does not contain any object code, but instead contains references to all public symbols of a DLL and are used by linkers to resolve external references to DLLs.

When linking a DLL, the built-in linker can automatically create an import library for it, and places it in the first directory mentioned in the **Library directories** option (in the Directories Options dialog).

When using an external linker to link a DLL, VP can run the IMPLIB utility supplied with VP to create an import library. IMPLIB can be used on any DLL, not just ones created using VP.

In the **Generate import library** field, choose the preferred method for creating an import library:

- Select **No** to prevent an import library from being created.
- Select **Use DLL** to create an import library based on the newly created DLL.
- Select **Use DEF** if a module definition file is to be used for creating the import library. This can be relevant if the entry points of the DLL should have names different from the names in the Pascal source code; please refer to the Language

Reference about Module Definition Files for details. As an alternative to creating a .DEF file, a **{&Linker}** directive can be used instead.

## Command-line tools

Virtual Pascal includes a number of tools for use from the command line, including the compiler and linker. Experienced users may want to use the command line compiler for automatically compiling with the desired settings. The Help generator and help converter can be useful to most users.

### The command line compiler VPC

The command line version of the compiler is called VPC.EXE. VPC itself is only a few kB in size, because the core functionality is stored in the compiler/linker DLL VPCOMP20.DLL, also used by the IDE.

To compile a program from the command line, start VPC with the desired options. The syntax is

```
vpc [options] filename [options]
```

or store these in a text file and call VPC with the name of this text file as a parameter (with an @ in front of the name), like this:

```
vpc @settings.cfg
```

VPC supports a number of command line options, and allows compiler directives to be supplied as well. The options supported are

Option	Description	Example
-A<unit=alias>	Specify unit aliases	-AWinTypes=Windows
-B	Build all units	-B
-C	Compile only (No link)	-C
-CO[:Dir:Cond]	Sets OS/2 as the target; enables the <b>OS2</b> conditional define.  If <i>:Dir:Cond</i> is specified, the 3-letter string substituted for %p is replaced with <i>Dir</i> . <i>Cond</i> is a list of conditional defines that are defined for this target.	-CO -CO:TST:TEST;XYZ
-CW[:Dir:Cond]	Sets Win32 as the target; enables the <b>Win32</b> conditional define.  If <i>:Dir:Cond</i> is specified, the 3-letter string substituted for %p is replaced with <i>Dir</i> . <i>Cond</i> is a list of conditional defines that are defined for this target.	-CW -CW:D32;DPMI;DOS
-D<syms>	Sets symbols listed as conditional defines	-DDebug;Test
-E<path>	Sets the <b>Exe output</b> directory,	-E\vp21\out.%P

## 52 Configuration and tools

---

Option	Description	Example
	where the .EXE, .DLL and .MAP file is stored	
-GD	Create <b>detailed</b> level map file	-GD
-GP	Create <b>publics</b> level map file	-GP
-GS	Create <b>segments</b> level map file	-GS
-I<paths>	Set <b>Include</b> directories where the compiler searches for files included using <b>{\$I}</b>	-I\vp21\source\rtl; \vp21\source\%p
-K<addr>	Set image base address (OS/2 DLLs only)	-K100000
-L<paths>	Set <b>Library</b> directories, where the compiler searches for import libraries	-L\vp21\lib.%p
-M	Only make modified units	-M
-O<path>	Set <b>Output</b> dir, where supporting files (.OBJ, .LIB, .VPI, .LNK, etc) are stored	-O\vp21\units.%p
-P[0   1   2]	Set compression level to 0 (None), 1 (ExePack 1), or 2 (ExePack 2). OS/2 target only.	-P1
-Q	Quiet compile. No progress is shown during compilation	-Q
-R<paths>	Set <b>Resource</b> directories, where the compiler searches for resource files included using <b>{\$R}</b>	-R\vp21\res.%p
-T	Generate <b>Location Information</b> . All units compiled in the <b>{&amp;LocInfo+}</b> state have location information included in the file when -T is specified.	-T
-U<paths>	Set <b>Unit</b> directories, where the compiler searches for units (either compiler .VPI and .LIB or .OBJ files or Pascal .PAS source files) included in <b>uses</b> clauses.	-U\vp21\units.%p
-V<.VPO file>	Use a .VPO options file compatible with the one used by the IDE for specifying settings for the compiler. This is a useful alternative to specifying a .CFG response file.	-V Project1.vpo

The following compiler directives can also be specified on the VPC command line or in the settings file. Please refer to the section on Compiler Options on page 43 and the Language reference manual for a complete description of compiler directives. Following

the table of compiler directives below, a sample VPC settings file (or *response file*) is shown.

Directive	Description
-\$AlignCode+	Code alignment <b>on/off</b>
-\$AlignData+	Data alignment <b>on/off</b>
-\$AlignRec-	Record field alignment <b>on/off</b>
-\$B-	Full boolean expression evaluation <b>on/off</b>
-\$CDecl-	Sets the C calling convention
-\$D+	Debug information <b>on/off</b>
-\$Delphi+	Delphi language extensions <b>on/off</b>
-\$Far16-	Sets the Far16 calling convention
-\$Frame+	Assembler routine stack frames <b>on/off</b>
-\$G3+	Sets the 80386 as primary target CPU
-\$G4-	Sets the i486 as primary target CPU
-\$G5-	Sets the Pentium as primary target CPU
-\$H-	Long strings <b>on/off</b> (When on, <b>string</b> = <i>AnsiString</i> )
-\$I-	I/O error checking <b>on/off</b>
-\$J+	Writeable <b>consts</b> <b>on/off</b>
-\$LocInfo+	Run-Time Location Information <b>on/off</b>
-\$M-	Run-Time Type Information <b>on/off</b>
-\$M16384,100000	Set the initial and maximum stack size in bytes
-\$Open32-	Open32 compatibility mode <b>on/off</b>
-\$Optimise+	Code optimisations <b>on/off</b>
-\$OrgName-	Original names (no unit identifier in front) <b>on/off</b>
-\$P-	Open string parameters <b>on/off</b>
-\$PMTType:<type>	Set the application type to <b>VIO</b> , <b>NOVIO</b> or <b>PM</b> .
-\$PureInt-	Pure Interface Unit <b>on/off</b>
-\$Q-	Overflow checking code <b>on/off</b>
-\$R-	Range checking code <b>on/off</b>
-\$S-	Stack checking code <b>on/off</b>
-\$SmartLink+	Smart Linking (.LIB file generation) <b>on/off</b>
-\$Speed+	Optimise for <b>speed</b> or <b>size</b>
-\$StdCall-	Sets the StdCall calling convention as default
-\$T-	Typed @ operator <b>on/off</b>

Directive	Description
-\$Use32-	Implicitly include the <i>Use32</i> unit <b>on/off</b>
-\$V+	Strict <b>var</b> string checking <b>on/off</b>
-\$W-	Always generate stack frame <b>on/off</b>
-\$X+	Enable extended syntax options <b>on/off</b>
-\$Z-	DWord-sized enumerated variables <b>on/off</b>
-\$Zd+	Include line number information in .OBJ/.LIB files <b>on/off</b>

### Sample VPC response file

The command line compiler can use a response file, typically given an extension of .CFG, instead of or in addition to specifying command line options.

A response file contains settings identical to those that can be specified on the command line, with each option starting on a new line of the file, for example:

```
/u\vp21\units.%p;\vp21\source\tv
/l\vp21\lib.%p
/r\vp21\source\rtl;\vp21\source\%p
/$SmartLink+
/$D-
/$Delphi+
/$Optimize+
/$W-
/$L-
/$V-
/$LocInfo+
/T
```

To make TEST.PAS using the options in the response file above, named TEST.CFG, the following command line should be used:

```
VPC TEST.PAS @TEST.CFG
```

### The Import Library manager IMPLIB

IMPLIB can be used to generate import libraries for DLLs, either by directly reading the DLL or by reading a .DEF file for it. Additionally, IMPLIB is able to generate a .DEF file listing all public entry points for a DLL.

To create a .DEF file for a DLL, the syntax of is:

```
implib [options] <name>.DEF <name>.DLL
```

This causes IMPLIB to analyse the DLL and create a .DEF file containing the name and description of the DLL along with a list of entry points exported by the DLL. The .DEF file created can then be used to create another import library for the same DLL, if so desired.

To create an import library (.LIB) from a DLL, the following syntax should be used:

```
implib [options] <name>.LIB <name>.DLL
```

- and to create an import library from a Module Definition File:

```
implib [options] <name>.LIB <name>.DEF
```

IMPLIB supports the following command line options:

Option	Description
/C	Generate case-sensitive library
/P#	Specify page size
/N	Import by name
/X	Force <b>NoName</b> attribute
/I	Process <b>IMPORTS</b> statements
/? or /H	Display help

IMPLIB can be run with a response file containing the desired parameters as well. To do this, specify the name of the response file on the command line, preceded by an “at” sign (@).

## The Help Generator HG

HG is used to convert Virtual Pascal .HLP source files to the format used for online help in the IDE, .VPH.

The syntax for running HG is

```
hg [options] HelpSourceFiles, Outputfile
```

HG supports the following command line options:

Option	Description
/C	Do not generate .VPH file; check for syntax only
/N	Create a file containing <b>const</b> declaration of topic numbers. This file is given the same name as the output file, with a .EQU extension.
/P	Pack (compress) help file
/W	Disable warning messages
/? or /H	Display help

The help source files are plain text files, divided into sections. To create links and index entries, several symbols are used as control characters. To include a character that is also a control character in the help, the symbol must be repeated twice.

Help source files can use the following control characters:

Symbol	ASCII	Meaning
	124	Treat the rest of the line as a comment
“Smiley face”	1	Highlight colour 1 (section must end with this symbol as well)
˘	96	Highlight colour 2 (section must end with this symbol as well)
“Dark face”	2	Mark an example (Can be copied to clipboard with the

## 56 Configuration and tools

---

Symbol	ASCII	Meaning
		Copy Example command in the IDE
~	126	Define a help topic. Syntax:  <i>~TopicName IndexString1 IndexString2</i>  where  <i>TopicName</i> is the name of the topic, <i>IndexString?</i> are the string(s) placed in the index
^	94	Define a hyperlink to a help topic. Syntax:  <i>^TopicName {Hyperlink text}</i>  The hyperlink text will be highlighted in the display of the help file, and clicking on it will navigate to the topic with the name <i>TopicName</i> .

A hyperlink can point to a topic anywhere in the current collection of help source files, but cannot point to a topic in another .VPH file.

### Example

A very simple help source file could look like this:

```
| Sample help file
~Sample Sample_Help
This is a `sample` help file. Click ^SomeTopic {here} to
go to another topic.

~SomeTopic Sample_Topic
This is another sample topic. To return to the first
topic, ^Sample {click here}.
```

To make a compressed .VPH file for use in the IDE from the above, run HG on the source file SAMPLE.HLP:

```
hg /P sample.hlp, sample.vph
```

Copy this file to the directory containing VP.EXE, or to the \Common subdirectory off the main install directory, and re-run the IDE to see the new help.

## The Help converter TPH2VPH

TPH2VPH (supplied as an OS/2 executable only) can be used to convert Borland Pascal help files with a .TPH extension to the Virtual Pascal .VPH file format containing help for use in the IDE.

The syntax of TPH2VPH is

```
tph2vph <name>
```

This causes the help file to be converted to Virtual Pascal Help file source and compiled to a .VPH file in one step. After the conversion, the help file can be made available for use in the IDE by copying the .VPH file to the \Common directory of the VP installation.

The .TPH extension of the file name should not be specified on the command line.



## **Error Messages**

Virtual Pascal generates two kinds of error messages: compiler error messages and run-time error messages.

A list of all error messages and their meanings can be found in the Language Reference and in the online documentation accessible from within the VP IDE by pressing F1.

# Debugging programs with VP

Programs containing more than a few lines of code almost always require a level of fine tuning before they work as desired. The built-in debugging facilities of Virtual Pascal provide a wide range of tools that can help the debugging process.

This chapter is intended to cover the basics of debugging, by introducing all of the different options available. Additionally a range of advanced debugging features available in VP is introduced at the end of the chapter.

## Why debug?

When writing a Pascal program the first problem is to get the compiler to accept your input, and to make it “understand” what you are trying to do. Even if it may sometimes be difficult to find out why a particular construct is not allowed by the compiler, it is not an insurmountable task since the Pascal language has a strictly defined syntax that can be looked up in the Language Reference Manual.

The compiler is also a good help in finding these syntactical errors. When there is a bug, the compiler immediately tells you where it is and what caused it. In the section labelled **Finding semantic errors** on page 58, the tools available to find and remove these bugs will be discussed.

Once the program has been successfully compiled, there might still be problems with it caused by flaws in the logic of syntactically correct code, and these flaws can be hard to find.

With this in mind, a large part of VP is centred around helping the programmer - you - find and remove these flaws in logic. Again, there are two kinds of logical flaws that are quite distinct:

- Flaws that cause the program to crash or otherwise misbehave, and
- Flaws that are not apparent; the program simply does not produce the desired results. This may or may not be obvious.

The main difference between the two types is that VP can tell you that the program contains the first kind of flaw, whereas the second kind requires you as the programmer to realise it yourself. Starting from the **Preparing to debug** section on page 61, we will look at the aids available for finding and removing both of these kinds of bugs.

## Finding semantic errors

Syntactical errors occur when trying to do something that is outside the Virtual Pascal language definition, or is otherwise illegal. The entire Pascal language as VP understands it is covered in the Language Reference section.

The easiest way to find syntactical errors is to instruct the compiler to compile your code by selecting **Compile/Make** or pressing the F9 key. If any syntax errors are discovered, the compiler opens the appropriate source file and positions the cursor to the error location. An error message will appear at the top of the screen, with a number and a brief description of the error.

To get more information about the error encountered, either look up the error code in the Language Reference or press F1 to display a detailed description of the error and the possible cause of the error in a help window.

In the course of removing the problem, you will most likely scroll up and down in the source file, open other source files, etc. If you need to quickly locate the original error position or remind yourself of the exact error code, you can press ^QW, which will do just that. Alternatively, you can select the **Search|Show last compiler error** menu item, which has the same effect.

When you have corrected the error, re-compile by pressing F9 again - eventually, your program should compile and link without error. If this is the stage you are at, but your program does not perform as expected, continue in the **Preparing to debug** section later in this chapter.

## Understanding compiler messages

A significant part of correcting code, which the compiler refuses to compile, consists of understanding the error messages it presents. Carefully reading the description of the error is a good start.

Some errors may not be obvious, or the compiler may seem to write a wrong error message. Consider the following code fragment, where *x* is assumed to have been declared as an *Integer* type previously:

```
if x = 2 then
  WriteLn('x is 2');
else
  WriteLn('x is not 2');
...
```

If you try to compile this, the compiler will move the cursor to the beginning of **else** and state *Error 113: Error in statement*. This may not look particularly helpful, unless you press F1 to get more help about the actual error. The online help enlightens us by stating that “*This symbol cannot start a statement. Most likely, you have put a semicolon before the else part of a conditional statement*”. According to the Pascal language as defined in the Language Reference section, it turns out that there should not be a semicolon before the **else** part of the **if** statement. Removing the semicolon helps, and the fragment compiles.

### A brief note on semicolons

While this is not meant to be a Pascal tutorial, semicolons *do* play a significant role in the Pascal language, and they are worth mentioning here, since the most frequently occurring mistake any aspiring Pascal programmer makes is one of misplacing or forgetting semicolons.

The important thing to keep in mind is that *Semicolons in Pascal act as statement separators, not statement terminators*. The **if...then...else...** statement shown above is *one* statement, and should not contain any semicolons, since there are no statements to separate. The final *WriteLn* has a semicolon after it, because more statements follow it in the source text (these are not shown).

### Type conflicts

Closely following semicolon errors, a common error is caused by a conflict between the formal parameters listed in the definition of a procedure or function, and the actual parameters with which it is called in your code.

## 60 Debugging programs with VP

---

Since VP is aimed strongly at porting 16-bit DOS and Windows code to 32-bit OS/2, this mistake is destined to be even more common when porting programs written in Borland Pascal.

The *Integer* type in VP is 16 bit, just as it is in Borland Pascal. However, using 16-bit values in a true 32-bit environment is inefficient and the compiler can generate much better code if 32-bit values are used. For efficiency, programs should include either *Use32* or *SysUtils* in the **uses** clauses of all units that form part of the program. Both of these redefine *Integer* to be identical to the 32-bit *LongInt* type.

*Use32* also redefines *Word* to be 32 bit. *SysUtils* does not do this, since the *Word* type is still 16 bit in Borland Delphi. Instead, the new 31-bit unsigned *Cardinal* type has been introduced in the *System* unit. See also the section on Integers, which discusses this subject in depth.

All units in the Run-Time Library of VP include *Use32*, and consequently use 32-bit *Integer* types wherever it is appropriate to do so. If you do not, you will experience type conflict errors that can be hard to understand. Consider the definition of the *GetFAttr* procedure as defined in the *Dos* unit:

```
procedure GetFAttr(var F; var Attr: Word);
```

Having looked at it, you might write the following program, using the *GetFAttr* procedure:

```
uses Dos;
var
  F      : Text;
  Attr  : Word;
begin
  ...
  GetFAttr( F, Attr );
end.
```

It does not work, however: a *type mismatch* error is reported. Having read the preceding section, the reason should be obvious: The *Word* type used in the definition in the *Dos* unit is 32-bit, and the *Word* used in the small program is 16 bit. The program can be made to compile by changing the **uses** clause to read

```
uses Use32, Dos;
```

instead, redefining the meaning of *Word* in the program. An alternative solution could be to simply change the definition of the *Attr* variable in the **var** section above to

```
var
  F      : Text;
  Attr  : LongInt;
```

While this would work, it potentially requires many code changes and it makes the code less readable. Not to mention that it will not work with Borland Pascal any more, where *Words* are still only 16 bit...

An alternative is to instruct the compiler to automatically include the *Use32* unit in the **uses** clause by enabling the **{&Use32+}** state.

### The compiler setup

Other compiler errors may be caused by wrong compiler settings. As described in Chapter 5 on page 21, the compiler supports a number of different language extensions and options that all affect the way the compiler interprets the source code.

The following small procedure:

```
function Test(ch: Char) : Integer;  
var  
    Result : Integer;  
begin  
    Result := ord(ch);  
    Test := Result * 2  
end;
```

compiles fine in the **{&Delphi-}** state, where *Result* does not have any special meaning, but fails to compile with *Error 4: Duplicate identifier 'Result' in the {&Delphi+}* state.

One way of avoiding this kind of error is to explicitly set the state of all relevant compiler directives at the beginning of each source file. VP can automatically convert all of the defined settings of the **Options|Compiler** dialog into the equivalent compiler directives and insert into the code; simply press ^OO to do this.

## Preparing to debug

While you are still working on a program, it is advantageous to tell the compiler to generate as much information about it as possible, to enable it to help you find possible errors.

You will also want the compiler to make as many “sanity checks” as possible, to help catch obvious errors in the code at an early stage.

To do this, go to the **Options|Compiler** dialog and click the **Debug** button. This will enable all of the compiler options that can help: Debug information, Range checking, Stack checking and Overflow checking. It also disables Optimisation, and enables the generation of stack frames for all procedures.

All of these options cause your executable program to become bigger and to run slightly slower, but enables you to find bugs that would otherwise be very difficult to find. A small price to pay, since all you have to do is click on **Fastest** once you (think you) have found and removed all bugs and then rebuild the program.

## Options | Debugger

Before moving on to the next session, the **Options|Debugger** dialog items will be briefly covered. If you are not an experienced user, you should go directly to the next session and return to this only when you are comfortable with debugging in VP. The options in this dialog determine the behaviour of the debugger, and include:

- **Display Swapping** can be either **None**, **Run Only** or **Smart**, and applies to situations where you debug the program by single-stepping through it. The name of the switch is inherited from the Dos environment and no longer describes accurately what it does:

When **None** is selected, the program does not accept keyboard or mouse input, unless you switch to the session manually using Alt-F5. The display is still updated correctly.

In the **Run Only** state, input is only accepted when **Run|Run** is selected.

In the default **Smart** state, input is only accepted when VP “thinks” there is a chance that this may happen during execution.

- The **Integer Format** radio button determines how integer type values are displayed in the Evaluator, Watch and Inspector windows. They can be displayed as **Decimal**, **Hex** or **Both**.

- The **Debugging** group contains two options:
  - **Debug Startup code**, which is disabled by default. When an OS/2 or Windows program starts executing, a host of operating system start-up code is executed before control is transferred to the program itself. This code is normally not of interest, but you can force VP to start the debugging session by debugging this code. If you do this, your debugging session will start in the *CPU Window* and point to somewhere in an operating system DLL. This option must be checked to debug DLL initialisation code. You will then be able to start the debugging session, place a breakpoint in the initialisation code of the DLL and run the program.
  - **Run in VIO Window** (OS/2 only) If this option is enabled (default), any text-mode VIO application will be run in a window on the OS/2 desktop when being debugged. If disabled, a text-mode application being debugged will run in a full-screen OS/2 session.
  - **Break on exception** determines what action the debugger takes when a *language* exception is raised. When checked, the debugger stops when an exception is raised, and displays a message containing the class and message of the exception. (Language exceptions are raised using the **raise** reserved word).
  - **Hard PM mode** (OS/2 VP/PM IDE only) determines which mode to use when debugging OS/2 Presentation Manager programs. When hard mode is enabled, the debugger is responsible for handling all PM messages, including those for other running tasks. This ensures that all messages are received in the same order as when the program is run outside the debugger. Please refer to the online help for further information about this setting.
- The **User Screen Delay** entry field should contain the number of 10ths of a second to delay before returning control to the debugger after pressing Alt-F5 to display the User Screen when debugging a PM application or a text-mode application in a window. The default value is 5 seconds (50 1/10ths). When debugging programs running in a Full-Screen session, control does not return to VP until a key is pressed. When debugging a program under Windows, pressing a key does not return focus to the debugger.

## Beginning the debug session

A debugging session can be started in a number of ways, depending on what you want to do. The simplest way is to press **^F9** or select **Run|Run**. If the program has changed since it was last compiled, it is recompiled and then started, otherwise VP will just run it.

After it has started, the word **Go** appears on the menu bar in white letters on a red background, and the IDE does not respond to mouse clicks or keyboard input.

You can however interact with your program or watch it executing. Once the program stops executing, either because of an error, or because it finished execution, control returns to the VP IDE. The reason why the program terminated along with an error code is displayed in a dialog box.

For more information about different methods of starting and continuing a debug session, please refer to the section on Stepping through the program on page 69.

## Starting and stopping the program

To temporarily stop program execution while it is running inside the debugger, press **Ctrl-Break** or click the **Stop** button on the VP/PM toolbar.

Once the program has been stopped, either by this method or by any of the other methods described later in this chapter, you can choose to either resume execution or stop the debugging session.

If you wish to resume execution either press ^F9, select the **Run|Run** menu, or click on the lightning-icon on the VP/PM toolbar.

If you do not wish program execution to continue, either press ^F2 to *restart* the program or Shift-F4 to *reset* it. ^F2 causes the program to be reloaded and make it ready to re-start execution. Shift-F4 causes all current debug information to be discarded and the current output window to close.

After a program has terminated, but before resetting or restarting the program, you can still examine both screen output and the values of static variables<sup>1</sup>.

## Run-time errors

If a run-time error was the cause of termination, a dialog box displaying the number and location of the error is displayed. When the OK button is pressed, VP displays the location of the error by opening the appropriate source file (if available), and positions the cursor on the statement that caused the error.

In most cases, knowing the exact location of the error will help you to quickly determine the cause of it. As for compiler errors, pressing ^QW takes you to the last error location, and redisplay the error message.

One problem with run-time errors, particularly range check, overflow and protection errors, is that the program terminates when the error is encountered. This makes it difficult to determine the state of the program when it failed, since you can no longer see the value of local variables, examine the call stack etc.

Two different methods can be of help in this case: Breakpoints and Exception trapping.

If the error occurs outside of the development environment, the error can be even harder to track, since the standard error message just outputs an error code and an address. To get more information in this case, please refer to the section about Run-time location information.

## Run-time location information

Run-time location information, RTLI, is generated by the compiler for every unit compiled in the **{&LocInfo+}** state. If the linker option to generate location information is enabled, the generated RTLI is included in the executable and can be used for error tracking purposes.

When a run-time error occurs in a unit that does not include RTLI, the error message can look something like

```
Runtime error 216 at 0001001C (Exception C0000005) TID=1.
```

This is useful if you have a map file – after all, you know that the code crashed in thread 1, and the address at which it crashed. However, if RTLI is available, the error message could look like this:

```
Runtime error 216 at 0001001C (STRINGS.PAS#781)
(Exception C0000005) TID=1.
```

---

<sup>1</sup> In Windows, the program window closes when the program terminates. To inspect screen output when the program terminates, place a *ReadLn* as the last statement of the program, or place a breakpoint on the last **end**.

## 64 Debugging programs with VP

---

Note, that this line contains information about the unit in which the error occurred (Strings.Pas) as well as the line number in which the error occurred (781).

Including RTLI adds about 1 byte of information per source line to the executable – RTLI has been implemented to make it as small as possible, at the expense of speed.

By calling the *GetLocationInfo* function defined in the *System* unit in your own exit procedure, it is possible to customise the error message displayed at run-time to suit your specific needs.

For example, the following small program does this:

```
var
  OldExit: Pointer;
procedure MyExitProc;
var
  FileName: ShortString;
  LineNumber: Longint;
begin
  ExitProc := OldExit;
  if GetLocationInfo(ErrorAddr, FileName, LineNumber) <>
  nil then
    WriteLn('Error ',ExitCode,' in ',FileName,
           ' line ',LineNumber)
  else
    WriteLn('Error ',ExitCode,' at ',
           Longint(ErrorAddr));
end;
var
  P: ^Longint;
begin
  OldExit := ExitProc;
  ExitProc := @MyExitProc;
  p := nil;
  p^ := 1;
end.
```

Since this program does little other than establish the exit procedure and then perform an illegal operation, running it outputs the following::

```
Error 216 in TEST.PAS line 25
```

If the program instead stops in the debugger and opens the CPU window, disable breaking on the Access Violation exception as described in the Exception trapping section – or run it from the command line for now.

## Breakpoints

The simplest form of breakpoint is a fixed location in the program, where the program is to be stopped whenever it reaches this location. Placing a breakpoint at or just before a place where a run-time error is known to have occurred allows you to examine the state of variables, the call stack etc. and can help you find the exact cause of the error.

A breakpoint is placed by moving the cursor to the source line where a break should occur, and press ^F8 or select **Debug|Toggle breakpoint**. The line now containing a breakpoint is highlighted in red, and execution will stop when the program reaches the breakpoint.

The breakpoint can be removed by moving the cursor to the breakpoint line and pressing ^F8 or selecting **Debug|Toggle breakpoint** again. To show a list of all breakpoints in the program, use the **View|Breakpoints** dialog.



If the program does not stop when it reaches the breakpoint, you probably have not enabled debug information. Make sure that you have followed the directions outlined in the Preparing to debug section.

Once program execution has stopped at a breakpoint, the tools described in the following section about Exception trapping can be used.

For information about Advanced breakpoint options, please to refer to page 70.

## Exception trapping

Run-time errors are often caused by CPU *exceptions*. An exception is an abnormal condition that causes an interruption of the normal flow of execution.

When the checking options (Range, Overflow and Stack checking) are enabled, code is included to generate an appropriate exception if one of the checks fail. In case of an invalid memory reference (General protection fault), which normally causes a run-time error 216 to be generated, an exception is also generated.

Normally, when an exception occurs, a run-time error is generated and the program terminates. Using the **View|Exceptions** dialog, this behaviour can be modified.

The dialog lists all operating system (OS/2 and Windows) exceptions of relevance, and a tick mark can be placed next to each exception either by clicking the checkbox, or by pressing the **Break** button. As the name of the button hints, placing a checkmark for a particular exception causes program execution to *break* instead of passing control to any exception handlers installed when this exception is encountered.

When an exception that is set to break occurs, VP opens either an *Editor Window* or the *CPU Window*, with the instruction that caused the exception to occur highlighted. (An editor window is only opened if the address of the exception matches the address of the beginning of a source line.) The state of the program is preserved, and can be examined as usual. If you are unfamiliar or uncomfortable with the *CPU Window* (described in detail on page 73), you can close it by pressing Alt-F3. If you then press Ctrl-Shift-O or select **Origin** from the local menu of an editor window (Press Alt-F10 or right-click the mouse to show the local menu), VP will position the cursor on the source line that caused the exception.

Enabling VP to break on exceptions is a very powerful way of quickly eliminating errors, since you do not have to know where the errors may occur before you execute the program.

The exceptions normally of interest are

XCPT_ACCESS_VIOLATION	An invalid memory address was accessed
XCPT_FLOAT_DIVIDE_BY_ZERO	Real number division by zero
XCPT_INTEGER_DIVIDE_BY_ZERO	Integer division by zero
XCPT_ARRAY_BOUNDS_EXCEEDED	Range check error

## Examining the state of the program

When program execution has been temporarily stopped, either because a breakpoint was hit, an exception caused a break, or by one of the methods described later in this chapter, it is often necessary to determine the state of the program, its variables etc.

### The Evaluator

Using the *Evaluator* dialog, it is possible to display the value of an expression and modify the value of variables and typed constants. Open the evaluator by pressing ^F4, or by selecting **Debug|Evaluate/modify**.

The dialog displays three input lines, labeled *Expression*, *Result* and *New value*. In the *Expression* field type the expression to evaluate. Enter the expression and press Enter, the expression is evaluated, and the result is displayed in the *Result* field. If the expression is a modifiable value, a new value can be entered in the *New value* field; press Enter again to change the value.

The evaluator supports complex expressions and format specifiers; please refer to the section on Format specifiers for further details.

### The Watch Window

The *Watch* Window can be used to display the values of a number of different expressions. To add an expression to the list of expressions to watch, press ^F7 or select the **Debug|Add watch** menu item.

In the dialog, enter the expression to watch (including any format specifiers), and press Enter. If the watch window is not open, it will be opened and will appear in the bottom of the screen.

An easy way to add an expression to the Watch window is by using the local menu for the editor window. Position the cursor to the expression and either right-click or press Alt-F10 to display the local menu. From this menu, select the **Watch** item, and the expression under the cursor will be added to the Watch window.

The Watch window also has a local menu that enables you to **Add** watch items, **Edit** items, **Remove** items, **Enable** items temporarily disabled, **Change** the value of an item and **Inspect** an item in an *Inspector window*. Options for enabling, disabling and removing all watches are also available in the local menu.

The Watch window supports complex expressions and format specifiers; please refer to the section on Format specifiers for further details.

If you wish to examine a CPU register, the Watch window supports this too. All register identifiers as well as two selector values, *FlatDS* and *FlatCS*, can be used as if they were defined in the *System* unit.

For example, you can watch either *CH* or *System.CH* to view the value of the CH CPU register. *Mem[esi]* or *Mem[FlatDS:esi]* displays the byte value at the address contained in the ESI register – in OS/2, this works even after the program has terminated.

### The Inspector

The *Inspector* is yet another method available for determining the state of your program and variables. The Inspector is particularly useful for inspecting complex structures such as arrays, records or objects - or a combination.

To inspect an expression, either select **Debug|Inspect**, or move the cursor in an editor window to the expression to be inspected and select **Inspect** from the local editor menu.

Unlike the *Watch window*, where all expressions being watched are displayed in one window with one line per expression, each expression being inspected requires its own *Inspector window*. Any number of inspector windows can be displayed simultaneously.

The top line in the Inspector window tells you where the expression being inspected is stored. This can be **Constant**, if you inspect a constant expression, a **memory address** displayed as @xxxxx, if it is a variable or typed constant expression, or it can be **Register**

if the expression being inspected is a local variable that is kept in a CPU register or if you explicitly specify the name of a CPU register for inspection.

Below the storage description, each member of the indicated data item is displayed together with their names, types and values. If the object is an array, a list of indices is displayed, with the values displayed on the right-hand side of the window. If the object is a record- or object-type, the list of identifiers together with their values is displayed.

Individual items in the array or object can be inspected by pressing Enter. This causes the view to zoom in on the current item, and the zooming in can continue until a simple type expression is encountered.

The Inspector window has a local menu, from where you can specify a **Range** to inspect (Array types only) or **Change** the value of an item in the list. For object types, it allows you to specify whether **methods** should be displayed along variable identifiers and whether **inherited** fields should be displayed. You can also **Inspect** the current data item (equivalent to pressing Enter), or you can **Descend** to a lower level without opening a new Inspector window.

Finally, the local menu allows you to inspect a **New expression**, **Type cast** the data currently being inspected to another type, or show the placement of the current data item in the **Object hierarchy** window (Objects and Classes only).

As for the Watch and Evaluate windows, the Inspector window supports format specifiers, described in the next section.

## Format specifiers

When watching or evaluating an integer expression, it is displayed in its default state as set up in the **Display format** radio group in the **Options|Debugger** dialog.

One or more *Format specifiers*, appended to the expression after a comma, can override the default display. For example, an *Integer* variable can be displayed as a memory dump, a *Pointer* can be displayed as a **string** - almost anything is possible - please refer to the following table for details.

In addition to applying format specifiers to expressions, typecasts can be used to change the format. After the list of format specifiers, a number of examples show some of the flexibility available.

Specifier	Effect
C	Character. Causes special display characters for control characters (ASCII 0 through 31) to be displayed. By default, such characters are shown using the equivalent #xx sequences. Affects only characters and strings.
D	Decimal. Shows all integer values in decimal. Affects simple integer expressions as well as arrays and structures containing integers.
Fn	Floating point. Shows n significant digits, where n should be an integer between 2 and 18. The default value of n is 11. Affects only floating-point values.
H,X,\$	Hexadecimal. Shows all integer values in hexadecimal with a \$ prefix. Affects simple integer expressions as well as arrays and structures containing integers.
M	Memory dump. Displays a memory dump, starting with the address of the indicated expression. The expression must be a construct that would be valid on the left side of an assignment

## 68 Debugging programs with VP

Specifier	Effect
	statement (a construct that denotes a memory address). If it is not, the M specifier is ignored.
N	As number. Causes Boolean and enumerated values to be displayed as numbers, rather than constant names.
P	Pointer. Causes pointers of the built-in type PChar to be displayed as pointers, instead of displaying the string being pointed to.
R	Record. Displays record or object field names, such as (X:1;Y:2) instead of (1,2). Affects only record and object variables.
S	String (default). Shows control characters (ASCII 0 through 31) as ASCII values using the #xx syntax. Since this is the default character and string display format, the S specifier is only useful in conjunction with the M specifier.

### Examples of format specifiers

Given the following declarations and code fragment:

```
var
  Coord : Record x,y : Byte; end;
  col   : (red, blue, green);
  I     : Longint;
  pS    : ^String;
  r     : Extended;
  pCH   : pChar;

begin
  Coord.x := 10;
  Coord.y := 20;
  col := green;
  New( pS );
  pS^ := 'Dynamic';
  r := pi/2;
  I := -$1234;
  pCh := StrNew( 'ABC' );
end.
```

If program execution is stopped after the last assignment operation, the following results could appear in the Watch or Evaluator windows:

Expression	Result	Expression	Result
Coord	(10,20)	SmallWord(Coord),xd	5130 (\$140A)
Coord,r	(X:10, Y:20)	Length(pS^)+Coord.x	17
Coord,rx	(X:\$A, y:\$14)	col	green
I	-4660	col,n	2
I,x	\$FFFFEDCC	r	1.5707963268
pCh	'ABC'	EAX,x	\$31004
pCh,p	Ptr(\$1B118)	r,f2	1.6
pS	Ptr(\$1B018)	i*r-25+ord(col),f4	-7342.9
pS^	'Dynamic'	I,2m	CC ED
pS,m	18 00 1B 00	I,md	204 237 255 255

---

## Stepping through the program

The **Run** menu features a number of different ways to execute a program a bit at a time.

If you place the cursor at an interesting location in the source code and press F4 (or select **Run|Go to cursor**), the program will be started and will not stop until execution reaches the cursor location (or another break condition, such as a breakpoint). **Go to cursor** can be considered to be a temporary breakpoint, which is only active until you set another temporary breakpoint or recompile.

Another way of executing a small portion of code is by *single stepping*. VP offers two different forms of single stepping: **Run|Trace into** (F7) and **Run|Step over** (F8). Both single stepping methods cause just a single line of source code to be executed. The difference between the two only becomes apparent when executing a statement containing a function, procedure or method call. In this case **Step Over** causes the procedure to execute, as if it was just another source line, whereas **Trace into** follows execution into the procedure call.

**Run|Execute to** (Shift-F9) prompts you for an address, after which the program executes until it reaches the address entered. This is equivalent to **Go to cursor**, except you specify an address instead of a cursor location to execute to. The address can take the form of any valid address expression, such as a number, a pointer or a procedure name.

**Run|Until return** (Alt-F8) causes execution to continue, until the currently executing subroutine is about to return to its caller.

### Run | Animate

The items in this menu allow single-step execution to take place automatically, until a key is pressed or the program terminates. For all the three animation options, you will be asked to enter an animation delay in 1/10th of a second units. If a value of 5 is entered, VP will wait 0.5 second before each single stepping command is executed.

- **Run|Animate|Trace into** (Alt-F4) is equivalent to having VP press F7 continuously.
- **Run|Animate|Step over** is equivalent to having VP press F8 continuously.
- **Run|Animate|Instruction trace** will open the *CPU Window* and single step each CPU instruction in the trace into mode. For more information about the *CPU Window*, please refer to the section on Advanced debugging.

**Run|Instruction trace** (Alt-F7) opens the *CPU Window* and executes a single CPU instruction. This is equivalent to pressing F7 if the CPU window is already the active window.

### Run | Parameters

Select this menu item to specify the parameters you wish to pass to the program being executed from the VP IDE. Parameters specified using this method are equivalent to parameters passed via the command line processor, if the program had been executed from there.

Please note that it is not possible to specify redirectors on the command line in the IDE (>, <, |). If these are specified as parameters to a program executed via the command line, they would have a special meaning because CMD.EXE treats them in a special way. If they are specified in the **Run|Parameters** in VP, they will be passed to the program as any other parameter.

Parameters passed to a VP program can be counted using the *ParamCount* function, and accessed using the *ParamStr* function.

### Advanced debugging

This chapter is meant for users with a working knowledge of the VP IDE and the topics covered in the last section. If you have reached a stage where you want to do something in the debugger that is not possible with the basic features explained so far, this is the place to look.

The core of the debugging features of VP can be accessed via the **View** and **Debug** menus. Each of those items that have not been covered in previous sections will be discussed in the following.

#### Advanced breakpoint options

Select **View|Breakpoints** to open a dialog containing a list of all breakpoints defined in your program. Each breakpoint is summarised on a single line in this window, and can be modified through the **Change Options** item on the local menu.

In this dialog each breakpoint can be configured in a number of ways:

- Every breakpoint has an address on which it triggers. This address can be entered as a line number of a source file, as the address of an identifier or as an absolute address expression.
- A breakpoint can either cause the program to **Break**, or it can cause a line to be added to the *Log window*. Or both. Select the desired actions in the **Action** field.
- Normal breakpoints are **Execution** breakpoints. When execution reaches the address specified, execution is interrupted. In addition to these, the 386 hardware supports up to 4 *Data points*, where action is taken when data at the specified address is read, written or accessed. Specify the type of breakpoint in the **Type** field.
- Breakpoints can be temporarily disabled. Check the **Enabled** box for those breakpoints that should be active.
- If you know that your program fails the 1018's time it reaches a certain location, you can specify a **Pass count** for the breakpoint. The default value is 0, meaning that the breakpoint is activated the first time its condition is met.
- In the **TID** field, you can specify a thread ID. If this is 0, the breakpoint will cause any thread meeting the break conditions to break, but you can specify that only a specific thread should break on a breakpoint.
- In the **Condition** field, you can enter any valid boolean expression in Pascal. Every time the breakpoint is hit, the **Condition** is evaluated, and the breakpoint is only activated if it evaluates to *True*. Note, that evaluating a condition every time can *severely* slow down execution.
- In the **Log expression** field, you can enter any number of Pascal expressions, separated by semicolons. If the **Log Action** is enabled, these expressions are evaluated and written to the *Log window* when the breakpoint is activated.

#### The Call stack

If stack frames are generated (**{\$W+}** state), the call stack can be viewed by selecting **View|Call stack** (^F3). This opens the *Stack Window*, which contains a list of all function, procedure or method calls that have brought execution to its current stage.

Each call is listed on one line, listing all parameters passed. On the local menu of the Stack window, you can select just one option, **Inspect**, equivalent to pressing Enter on the

currently highlighted item. This moves the cursor to the point just after where the call was made, and where execution will return to once the subroutine has finished execution.

## The Log window

During a debug session, VP constantly updates the *Log window* with information about module loads, threads starting and stopping, **Log** breakpoints and user messages.

The local menu of the *Log window* allows you to **Open Log file** if you wish to store the content of the window in a file as well as display it in the window. When you are done with a particular log file you can **Close log file**.

Select the **Logging** option to toggle logging on/off. Select **Add comment** to add any text string (you will be prompted for it) to the log, and select **Erase** to empty the current log window.

## The Symbols window

Use **View|Symbols** to open the *Symbols Window*. This window is a dual-pane window displaying information about all symbols in your program. In the top pane, global symbols are listed, and in the bottom pane local symbols can be found.

By default, all variables, procedures and symbols declared in the **interface** part of any unit, as well as declarations from the **implementation** part of the current unit, are displayed in the top window, and all local variables defined in the current procedure, function or method are displayed in the bottom window. All symbols in the top window are fully qualified by unit name to allow for duplicate identifiers across multiple units.

The Symbol window can be set up either by the **Options|Symbols** menu or by selecting **Options** from the local menu. In this dialog, you can choose to include labels, constants or type definitions to the information already displayed. In the **Display** checkboxes, you can enable displaying not just the values of variables, but also the types. You can also choose to have all parameters listed for procedures, functions and methods listed in the top pane by checking the **Show procedure parms** checkbox.

The Symbols window supports incremental searches in both panes, to allow you to quickly locate a particular symbol. If you do not know the unit in which the symbol is defined, start your incremental search by typing a full stop (.), which is used to separate the unit name from the names of identifiers. As you type more characters, the cursor moves to the right and moves the cursor line to point to the first matching item.

## The Unit list

To get a list of all units and include files in the current program, select **View|Units** or press Shift-F3. This opens a dialog, listing all units currently included in the project.

All units compiled with *debug information* enabled have a (+) sign to the right of their name.

The unit list also displays files included using the **\$I** directive, and supports incremental search (type a partial unit name to find it quickly).

Move the cursor to a unit and press Enter, to load that source file into an editor window.

## The Threads list

Select **View|Threads** or press F6 to get a list of all currently active threads. The *Threads List* opens, displaying the list of threads along with their current status and location.

## 72 Debugging programs with VP

---

If you want to follow the execution of a single thread, and be sure that all other threads are inactive, you can use the **State** button to *Freeze* or *Thaw* each individual thread in the process.

### The Object Hierarchy window

VP can display the currently defined hierarchy of objects in an outline style format. Select **View|Hierarchy** to open the *Hierarchy window*.

If you press Enter on a particular object, or select the **Inspect** item from the local menu, an *Inspector window* is opened for the selected object. Note that the inspector is inspecting the object *type*, not a particular instance of the object.

### The Dump window

To get a formatted dump of a particular memory location, select **View|Dump**. This opens a *Dump window*, displaying the content of memory at the desired location.

If a question mark (?) is displayed instead of data, your program does not have access rights to the memory being viewed; accessing this memory from your program would cause an access violation exception.

Any number of Dump windows can be opened at any one time.

The local menu allows a number of different actions to be taken:

- **Go to** lets you enter any valid address exception, and the Dump window will display this portion of memory.
- **Follow** should be used to interpret data at the current position as a pointer that should be followed; it can be interpreted in a number of ways: **Near code** should be used if the data at the current cursor position can be interpreted as a near 32-bit pointer to executable code. The *CPU Window* is opened at this location. **Far code** is when the data is a 16:16 segmented pointer to executable code. Again, the *CPU Window* is opened at the location specified.

**Offset to data** causes the current memory location to be interpreted as a flat 32-bit memory location, and the dump window moves to display this portion of memory. **Segment:Offset to data** is interpreted as a 16:16 bit pointer to data, and **Base segment:0 to data** interprets the 16-bit word as a segment selector, and displays the memory at this segment:0.

- **Previous** restores the dump window to display memory at the location it showed before the last change.
- **Change** is used to change data. You will be prompted for a new value (it is possible to enter a list of values separated by spaces as well).
- **Display as** is used to specify the display format used and can be changed to any of the VP simple integer or floating point types. For example, selecting *Extended*, causes the data to be displayed as 10-byte FPU floating point values. When *Byte* is selected (the default), data is displayed as single hexadecimal bytes *and* as a character dump on the right hand side of the window
- **Marker** sets a “bookmark” at the current memory location. Moving the cursor causes the relative offset to the marker location to be displayed at the top of the window.
- When **Fixup** is selected, an address expression must be entered. This can be a CPU register (eax, for example) or can be an expression involving Pascal variables or constants. After pressing Enter, the Dump window will evaluate the Fixup expression



and go to that address. This can be very useful when debugging; setting the fixup expression to `esi`, for example, means that the data at the address pointed to by the `esi` register is always available at a glance - even when the value of the register changes.

- Select **Remove fixup** to clear the fixup expression for the dump window and return to normal operation.
- The **Filter** item acts as a toggle control. When filtering is disabled, all characters in the 32-255 range are displayed in the ASCII dump to the right of the window. When filtering is enabled, only letters, numbers and symbols are displayed; all others are displayed as dots (.).
- The **Block** menu allows you to manipulate memory directly. You can **Clear** a range of memory (fill it with 0s), you can **Move** a range of memory or you can **Set** a range of memory to a specific value.

The **Block|Read** option allows you to fill a block of memory with data read from a file, and the **Block|Write** command writes a range of memory to a disk file.

- Using the **Search** item, you can search for a set of bytes in memory. Use the **Next** item to repeat the search.

See also the CPU window, which includes a dump window.

## The Registers window

Open the *Registers Window* by selecting **View|Registers**. This window displays the values of all CPU registers, selectors and flag bits.

Through the local menu, it is possible to **Increment**, **Decrement**, **Zero** or **Change** the value of any register. Pressing Enter has the same effect as selecting **Change**.

When a register has changed, it is highlighted in white. This makes it easy to quickly see which registers are changed by executing a statement or an assembler instruction in the *CPU Window*. The colours of the registers change every time the program has executed using any of the means available (F4, F7, F8, ^F9, etc.).

See also the CPU window, which includes a registers window.

## The NCP window

VP allows you to display and modify the current state of the Floating-Point Unit (FPU), or Numeric Co-Processor (NCP). Select **View|Numeric coprocessor** to open this window.

The local menu in the *NCP window* allows you to **Zero**, **Empty**, or **Change** the value of any of the values on the NCP stack. If you are unfamiliar with the way FPUs work, please refer to Intel technical documentation.

The *Status Word* and *Control Word* panes have separate local menus, which allow you to either **Toggle** the state of the words, or explicitly set a new value with the **Change** menu item. For a description of the NCP status and control words, please refer to Intel technical documentation.

## The CPU window

The *CPU Window* provides instant assembler-level access to your programs. It contains a combination of the *Registers* and *Dump* windows, plus a Stack display and a disassembly pane containing information about both the source code and the resulting assembler instructions which the compiler generated based on the source code.

## 74 Debugging programs with VP

---

It is probably the most powerful debugging feature of VP, and can even act as a stand-alone debugger for any 16- or 32-bit OS/2 executable programs and 32-bit Windows programs.

Open the *CPU Window* by selecting the **View|CPU** menu item. The functionality of the Dump window and Registers window have already been described, so this section only contains information about the possibilities of the CPU and the Stack panes.

In the CPU pane, low-level information about your program is displayed:

- Each source line is displayed (if source code is available, and Debug information is turned on with **{SD+}**) preceded by the unit name and line number of the source line.
- Following the source line, the resulting machine op-codes and corresponding assembler instruction mnemonics are displayed.

For a function containing a simple loop, for calculating the sum of the elements of an array, like this:

```
function GetSum: Longint;  
var  
    i      : Longint;  
    Sum   : Longint;  
begin  
    Sum := 0;  
    for i := 1 to 10 do  
        Sum := Sum + x[i];  
    GetSum := Sum;  
end;
```

the output in the *CPU Window* could look like this (In the **{&Optimise+}** state):

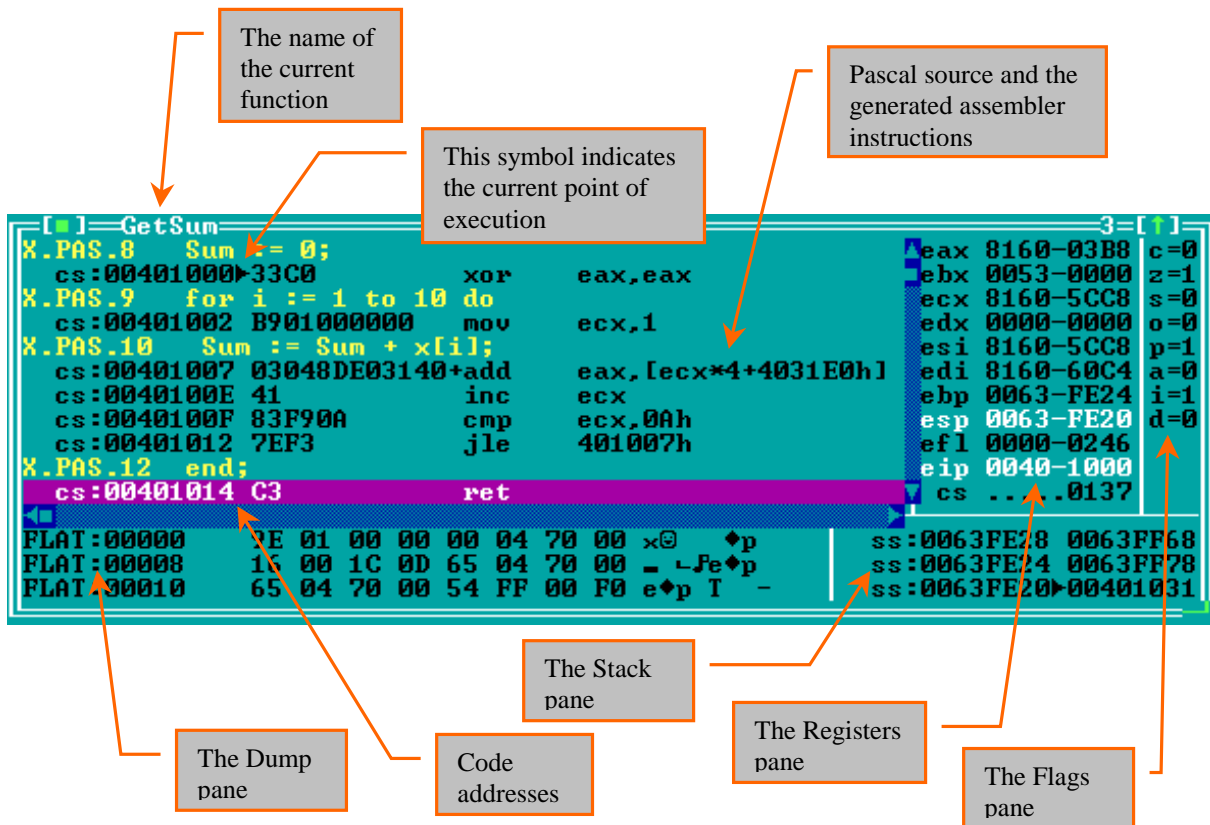


Figure 4: The CPU Window

The *Sum* variable is kept in the *eax* register, and the value of *i* is kept in *ecx*. The base address of the *x* array is 4031E0h - if you are proficient in Intel assembler code, the rest should be obvious. Nevertheless, here is a quick reminder: *xor* a register with itself to efficiently zero it; *mov* assigns a value (to a register, *ecx*, in this case); *add* adds a value to *eax*; *inc* increases it by 1, *cmp* compares it with a value, *jle* jumps to the specified address if the value was “less than or even”, and *ret* returns to the caller.

The local menu of the CPU window allows the code and registers to be manipulated, and provides the following options:

- **Goto...** prompts for a valid address expression, and moves the view to that address.
- **Origin** moves the view to the current *cs:eip* address.
- **Follow** follows a *call* or *jmp* instruction to its destination, without actually executing the code. Even indirect calls and jumps via a register or memory can be followed.
- **Previous** reverts the display to the last location, before the last **Origin**, **Follow** or **Goto** command. VP stores the last five locations, so the **Previous** command can be used to backtrack long sequences of **Goto** and **Follow** commands.
- Use **Search** to search for a sequence of bytes or a particular assembler instruction. VP uses the built-in assembler to assemble any instructions you type, and searches for the resulting sequence of bytes. Use **Next** to repeat the search.

## 76 Debugging programs with VP

---

Note, that code generated for short near calls and jumps, the instruction generated depends on the *address* where it is assembled. Do not try to such instructions.

- **View Source** is used to revert to the source display for the code currently being displayed in the CPU window.
- The **Mixed** item toggles between three different display modes. Select **No** to see only the disassembled instructions, **Yes** to see the default view exemplified above, where source code and assembler instructions are interspersed, or select **Both** to have source lines replace the assembler instruction lines where appropriate. This is very useful when debugging code written in the built-in assembler.
- Use **New CS:eIP** to set a new address from which execution will be resumed. A very useful function that should be used with appropriate care! It is particularly useful, if an instruction causes an exception, and you wish to disregard the offending instruction and continue execution after it.
- The **Assemble...** command allows you to enter any assembler instruction as you would do it in an **asm..end** block in your Pascal source code. The instruction entered will be compiled and replaces any instructions already present at the current cursor location.

While this is a powerful tool for dynamically changing and testing various instructions, it should be used with great care. Also note, that any changes made to the program being debugged are lost the next time you compile, load or reload the program from disk.

### The CPU window as a stand-alone debugger

The CPU window can be used to debug stand-alone executables.

In OS/2, any 16- or 32-bit OS/2 executable can be loaded; in Windows, any native Win32 application can be debugged. To load the executable into the VP IDE, select **File|Load program...**, and enter the file name of the executable.

When debugging stand-alone executables, no debug information is available. No source tracking and variable inspection is possible, but all other features are available.

In OS/2, the debugger supports standard .SYM files. .SYM files for all of the standard OS/2 DLLs are available from IBM, and include in these the names of many DLL entry points. Using IBM's MAPSYM utility, the .MAP files generated by VP can be used to produce .SYM files.

To specify the location of .SYM files, use a /F<path> parameter on the command line when specifying the IDE. When debugging modules for which .SYM files are available, VP will show the names of the entry points in place of addresses in *call* and *jmp* statements.

## VP example programs

VP comes with a selection of example programs, designed to assist in overcoming the initial hurdles when programming using VP.

The examples are divided into 3 groups: Common, OS/2-specific and Win32-specific. In addition to the examples supplied with Virtual Pascal, a number of examples and libraries are available for download from the fPrint FTP site as well as from a number of third-party sites.

### Common Virtual Pascal examples

These examples reside in a directory off the Examples subdirectory, and illustrate many different aspects of programming in VP.

- *Delphi*. This directory contains 4 examples, demonstrating the use of some of the Delphi-compatible units included with Virtual Pascal:
  - *ClsDemo* demonstrates some of the capabilities of the *Classes* unit, the use of **is** and **as**, *TStringList*, *TBits*, *TCollection*, *TFileStream* and *Tparser*
  - *SysDemo* demonstrates some of the *SysUtils* unit, using the system settings, the *Format* functions, conversion functions, the *Currency* type, date manipulation functions, string functions and disk/file functions.
  - *TlsDemo* demonstrates the use of thread local storage using **threadvar** declarations in a multi-threaded program,
  - *XcptDemo* demonstrates the use of **try..except** blocks, i.e. Delphi-style exception handling.
- *Dll*. Shows how to create a DLL (*TstVpDll*), and two different ways of using a DLL (*TestDll1.pas* and *TestDll2.pas*).
- *MemMgr*. The main file in this directory is *MemMgrEx.pas*, which uses a sample memory manager installed using the new *TmemoryManager* interface. The installed memory manager (*PerfMgr.pas*) does not change the memory allocation scheme, but instead measures the time spent in the memory allocation calls. In order to measure the time very accurately and introduce as little overhead as possible, the timing is performed using the Pentium-only RDTSC instruction.
- *RatRace*. Demonstrates how to program a simple multithreading application in VP. The multithreaded application controls access to critical routines by using a Mutex semaphore, initialised through a *VpSysLow* function. The program initialises 8 “rats” in different colours that are assigned a thread each. Every time a thread becomes active, the rat moves forward one character. The final positions of the rats are written at the bottom of the screen.
- *Sort*. Shows an example of a highly optimised algorithm for sorting text files, using a mix of Pascal and 32-bit assembler code.
- *Test*. This directory contains 4 examples, demonstrating compatibility with the *Crt*, *WinCrt*, *Dos* and *WinDos* units of Borland Pascal:
  - *TestCrt* demonstrates compatibility with the *Crt* unit by playing a little tune, while filling a text window with a string of various colours. When a key is pressed, it

manipulates the content of the window. The second part displays a "star sky", with blinking stars.

- *TestDos* and *TestWDos* demonstrate compatibility with the *Dos* and *WinDos* units: searching for files, expanding file names, etc.
- *TestWCrt* shows the use of the *WinCrt* unit. It simply writes some text to the window, and allows the user to enter text. Terminate the example by typing q and Enter.

These programs can also be compiled with Borland Pascal without any changes.

- *TestUtil* demonstrates some of the abilities of the *VPUtils* unit by showing various system and current settings: Current time, process ID, keyboard state, window size, boot drive, drive type, code page, dynamic memory in use, etc.
- *Touch* implements a simple Touch program, changing files' date and time to the current one.
- *UnRAR*. Implementing a fully functional Pascal version of an unpacker for the popular *RAR* archiver, which is the compression program also used for compressing the VP distribution archives. The code is a direct port of the public C source code version of *UnRar* - this also is evident from the Pascal code.

## Borland Pascal examples

Borland Pascal comes with a great number of examples, most of which can be immediately compiled with Virtual Pascal. The three most interesting ones do require a number of small changes to the source code.

These examples all reside in the `EXAMPLES\COMMON\BP7` directory and subdirectories, and `.DIF` files outlining the changes to make to the original source code. You must already have the Borland Pascal source code in order to be able to compile these examples. Once the changes are applied, the examples will compile for either a Win32 or an OS/2 target.

The changes required are detailed in a text file; one per example:

- *BreakOut*. This example, which implements a text mode version of a classic game, can be found in the `BREAKOUT` directory. The changes required to make the program work are listed in the `BREAKOUT.DIF` text file in the same directory.
- *TVDemo*. Residing in the `TVDEMO` directory, this example demonstrates a range of features available to programs using the Turbo Vision library from Borland. The changes required to the source code of this example are detailed in `TVDEMO.DIF`.
- *TVFM*. A powerful program also written using Turbo Vision, implementing a text-based File Manager clone. With multiple windows support, drag and drop and even a trashcan, it is a powerful demonstration of the capabilities of Turbo Vision. Changes required to the source code are listed in `TVFM.DIF`.

## Windows-specific examples

These examples reside in the `W32` directory off the Examples subdirectory. In this version of Virtual Pascal, the only Win32-specific examples are examples demonstrating the OWL compatibility of VP.

These examples are installed as source code updates of the Equivalent Borland Pascal for Windows examples.

- *Calc* demonstrates a tiny OWL program that implements a simple calculator.
- *OwlChess* is a full-blown chess program, that is able to show a chess board as well as be your opponent in a game of chess. Please make sure **OWL** is defined and that the RTL is re-built before compiling this example.

## OS/2-specific examples

These examples reside in the OS2 directory off the Examples subdirectory, and illustrate some of the aspects of OS/2 programming using Virtual Pascal.

- *Clock*. The Clock example is an example of an OS/2 Presentation Manager program. It implements a very simple analog clock, which when minimized continues to update its minimized icon.
- *Dive*. The example, *TestDive.Pas*, illustrates how to use the DIVE wrapper class implemented in the *Os2Dive* unit. DIVE is OS/2's Direct Video Extensions API.
- *EA*: An examples showing how to use and access OS/2 Extended Attributes, *TestEA*. This example uses the *tEAList* class implemented in the *Os2EA* unit, and uses this to manipulate the list of extended attributes of the file specified as a parameter on the command line.
- *ObjClass*. This directory contains two examples:
  - *ListCls* shows how to enumerate objects registered with the OS/2 Workplace Shell.
  - *NewObj* demonstrates how to create a program object on the OS/2 desktop using VP. First, it forces an error and demonstrates how to obtain extended error information from OS/2. It then creates a "VP Test Object" which when opened will attempt to load C:\CONFIG.SYS into E.EXE.
- *Os2Exec*. This directory contains an example, *TestExec*, showing how to use the *TredirExec* class implemented in the *Os2Exec* unit. This class allows you to execute a child process and capture the output from that process; the example uses this to issue a DIR command, showing the output in a window.
- *Rexx*. This directory contains examples showing how to interface with the REXX command processor of OS/2, in two examples:
  - *CallRexx*. Shows how REXX functions can be called directly from Virtual Pascal. When run, it executes two REXX scripts, the first of which plays a little tune, the second of which TYPEs C:\AUTOEXEC.BAT.
  - *RexxExt*. Demonstrates how to write a DLL extending the functionality of REXX in VP by implementing a REXX function called *VPtouch*. Includes a small REXX script demonstrating how to use the DLL, *TestExt.Cmd*.
- *Flame*. Demonstrates how to do full-screen graphics in OS/2 using VP. The program runs in full screen mode only. If you try to switch to another session using Ctrl-ESC or Alt-ESC while *Flame* is running, it may appear as if OS/2 hangs. What really happens is, that OS/2 delays taskswitching commands for 30 seconds while full screen graphics are running. *Flame* is terminated by pressing any key.
- *SysLevel* is a simple program implementing roughly the same functionality as OS/2's SysLevel command, but running significantly faster. Shows how to efficiently implement *FindFirst/FindNext* calls to enumerate all files on a hard disk drive.

## 80 VP example programs

---

- *Triplex*. Is an OS/2 Presentation Manager game, somewhat similar to the popular TETRIS game. The purpose of the example is to demonstrate the steps involved in implementing a PM program using VP.



## Extending Virtual Pascal

Virtual Pascal v2.1 supports two major platforms, OS/2 and Win32, and includes a debugger for natively debugging programs running on both platforms.

In addition to this, it is possible to extend the functionality of both the compiler and the debugger to include support for more platforms. Doing so involves several steps: RTL support, linker support, and debugger support.

This work has started for the Linux v2.x platform, where some preliminary support is included in this release.

These steps are outlined in the following:

### Interface to the OS

The first step is to interface to the operating system API. This should be done by converting header files (probably from C) to Pascal, listing every API call in the operating system, types, etc. For Windows, the *Windows* unit holds this information; for OS/2, the *Os2Def*, *Os2Base*, *Os2MM*, etc. units holds it.

### RTL support

The Run-Time Library is platform-independent, with all platform dependencies extracted and localised in the *VpSysLow* unit. Enabling VP to generate object code for platforms other than OS/2 and Win32 is, at least in theory, a matter of porting the *VpSysLow* unit to the new operating system, using the API calls made available previously.

For the most part, this should be a straightforward job in practice as well. However, some parts may prove difficult to do, because the *System* unit (which uses the *VpSysLow* unit) relies on certain operating system characteristics, particularly in the area of exception handling. The structures passed in OS/2 and Win32 are similar, and it may be difficult to make the port work if the exception handling mechanism is very different from the model currently used.

For such systems, the *System* unit will require some customisation, implemented using conditional compilation. Please refer to the *System* unit source code for details on how this is currently done.

When the *VpSysLow* unit has been ported to another operating system, such as Linux or BeOS, the compiler and IDE should be just a recompile away from being able to run on those systems as well.

### Linker support

The next phase is to implement linker support. VP's built-in linker can use OMF standard object files and libraries (.OBJ and .LIB) and create both OS/2 LX and Win32 PE executables from these.

If the target operating system supports one of these executable file formats, the built-in linker can be used to produce executables for the operating system, and the system is now useable.

This is, for example, the case for certain 32-bit DOS extenders, which are implemented to support the Win32 PE executable file format as well as a subset of the Win32 API. On such extenders, such as RTX-DOS/32, programs compiled and linked with VP for a Win32 target can run with no changes.

If the operating system does not support any of these executable file formats, a linker native to the operating system must be used. If such a linker exists, and is able to read and process OMF style object files, it is simply a matter of configuring the VP IDE to use this linker, if it runs on either Win32 or OS/2. If it does not run on Win32 or OS/2, the link process can take place directly on the target platform instead.

If a linker exists, but does not support OMF-style object files, a new linker needs to be written. Writing a linker is a complex task, and this may prove to be the most difficult stumbling block in making VP produce executables for other operating systems. Please refer to any technical documentation on the operating system, file formats, and linkers that can be found for more information.

## Debugger support

Once the above steps have been completed, VP is able to produce executables for a new operating system, but is not able to debug them.

Fortunately, the debugger VP is based around an Open Debug API: By using the functionality provided by the ODAPI, it is possible to write support for debugging programs running on other platforms, across network connections, or even direct cable connections.

In order to provide debugger support for the new platform, or to change the behaviour of the existing debugger, the set of functions comprising the ODAPI must be implemented as a DLL, to be loaded by the Virtual Pascal IDE when debugging applications.

The interface to be supported is defined in the Pascal source file *VpDbgApi.pas*, located in the Source\Rtl\Debug directory. This unit shows how debugger support is implemented for OS/2 and Win32 and can be used as a template for extending the functionality to other platforms.

Please refer to the online help and Appendix C of the Language Reference Manual for a full description of ODAPI, its entry points, and the way in which it can be integrated into the VP IDE.

## Defining new targets in the IDE

The IDE supports the notion of targets other than the default Win32 and OS/2 ones. When a new target platform has been implemented as outlined above, or if you just would like to have a range of specialised targets, the IDE can be customised to make these targets selectable from the Compiler -> Target Platform menu item.

To define additional targets, edit the VP.INI file located in the \Common directory of the Virtual Pascal installation.

In this file, a single section defines the list of additional targets available for selection, separated by semicolons. The properties of each target is defined in a separate section per target. For example, to define three new targets called OWL, DPML and Linux, add the following to VP.INI:

```
[Targets]
Targets=OWL;DPML;Linux
```

```

[OWL]
Platform=W
Subst=OWL
Defines=WINDOWS;OWL;WIN32

[DPMI]
Platform=O
Subst=DOS
Defines=DOS32;OS2;DPMI

[Linux]
Platform=W
Subst=Lnx
Defines=Linux
ExecCustom=\vp21\bin.w32\pe2elf.exe
ExecParm=%s

[DriveSubst_Linux]
Count=2
Subst_1=P:~/home/am
Subst_2=T:\vp21~/home/am/vp21

[DriveSubst_Os/2]
Count=2
Subst_1=P:~C:
Subst_2=T:\vp21~C:\sources

```

After applying any changes to VP.INI, restart the IDE to make the changes take effect. In the above examples, 3 new targets would be available for selection in the IDE.

The OWL target would, when selected, use the string “OWL” in %P string substitutions in directory names etc, cause the conditional defines WINDOWS, OWL and WIN32 to be defined, and cause the compiler to link the program as a Win32 executable.

The DPMI target would use “DOS” as the %P substitution string, define DOS32, OS2 and DPMI, and would cause the compiler to link the program as an OS/2 executable.

The Linux target would use “LNX” as the %P substitution string, define LINUX, and create a PE executable. After the link process, a program called pe2elf.exe would be executed, with the fully qualified name of the executable as the parameter.

The “DriveSubst” sections are used when working in environments that share source code but where drive letters change – or are not used, as is the case in Linux. The compiler looks for a DriveSubst section matching the current execution environment (i.e. is independent of the target selected) and reads the list of substitution strings specified. For each drive substitution string, the string to the left of the question mark character (?) is substituted by the string to the right of it, for all parts of every search path used by the compiler.

For example, if you are running in OS/2, and your Units search path is

```
p:\sources\vp;t:\vp21\source\rtl;c:\test
```

the translated source path used when compiling the program (to any target) will be

```
c:\sources\vp;c:\sources\source\rtl;c:\test
```

This is particularly useful if the source code or other paths are shared on a network, or if Virtual Pascal is run in an emulated environment such as VMWare.

## Restrictions in target definitions

The settings specified in VP.INI must adhere to the following restrictions:

- The *Platform* setting can be either **W** or **O**, representing Windows and OS/2, respectively. When OS/2 is specified, the OS/2 Import Library OS2.LIB is used and OS/2 LX executables are produced. When Windows is specified, the Win32 Import Library IMPORT32.LIB is used, and Win32 PE executables are produced.
- The *Subst* string is used to replace any occurrences of the string %P in any unit, library, resource, output or include paths specified. This means that it is possible to separate both source code and output files for different platforms into separate directory structures; it also means that you should make copies of the standard import libraries (located in the LIB.OS2 and LIB.W32 directories) if your new target does not provide its own import library.
- The *Defines* setting can be up to 80 characters in length, and can contain any number of valid conditional define identifiers, separated by semicolons.
- The *ExecCustom* setting should be blank, or contain the name of an executable program. If defined, this program will be executed with the content of *ExecParm* as its parameters, after the normal VP compile and link process finishes.
- The *ExecParm* setting should contain any command line parameter to pass to the *ExecCustom* program, substituting the first occurrence of %s for the name of the executable just created.
- The *DriveSubst* sections that the compiler looks for are “DriveSubst\_” followed by either Linux, DPMI, OS/2, Win32s, Win9x or WinNT – the strings returned by the SysPlatformName function in VpSysLow. The paths substituted using the strings here are all normal search paths: Output, Exe Output, Units, Include, Resource and Library directory search paths.

## Using the Command-Line Compiler

The command line compiler can be called directly, without using the IDE, while retaining the ability to specify additional target information. This is done by an extension to the /Cx command line option, which allows the information specified in VP.INI for the IDE to be passed on the command line.

The syntax when compiling for Windows is

```
VPC . EXE /CW[ :Subst :Defines ]
```

For OS/2 targets, the syntax is

```
VPC . EXE /CO[ :Subst :Defines ]
```

If /CW or /CO is specified by itself, the default Windows or OS/2 targets are used. To specify a new target, such as the OWL target from the example above, use the additional parameters to specify the substitution string and conditional defines:

```
VPC . EXE /CW : OWL : WINDOWS ; OWL ; WIN32
```

Note, that the drive substitutions specified in VP.INI for the IDE are also read and used by the compiler, depending on the platform on which it runs.

## Win32 Keyboard Support

During the extensive Beta program of Virtual Pascal, the biggest problem was implementing good keyboard support for Win32. The keyboard handling functions used

by all programs, implemented in the platform-specific *VpSysLow* unit, proved to be difficult to get working in all conditions. In particular, supporting the very varied keyboard layouts used throughout the world proved difficult.

For this reason, the default keyboard handling code is located in a unit called *VpKbdW32*; the source code can be found in the `\Source\W32` directory. In most programs, this should not be of concern as this code is linked statically into your program and is used by the *VpSysLow* functions.

If, however, your program (or the VP IDE itself) exhibits strange behaviour related to the keyboard, and you wish to change this behaviour, VP supports this.

On startup, all Win32 programs written with VP, will look for a DLL called *VpKbdW32.Dll*. If this DLL is found, and defines the required entry points, the keyboard handling functions defined in this DLL will replace the default keyboard functions linked into the program.

To create a modified keyboard handler, load the *VpKbdW32.Pas* unit into the Virtual Pascal IDE, and define the conditional define **KEYDLL**. Save the unit under a new name, and modify this file according to your particular needs. Compile the file, and a VP Keyboard DLL is the result.

The interface to the DLL is quite simple, and consists of two entry points. The first entry point, *KbdInit*, initialises the DLL with pointers to variables used by the keyboard handler and *VpSysLow*. The second entry point, *KbdUpdateEventQueues*, is called regularly by functions in *VpSysLow*, and should cause the mouse and keyboard event queues to be updated with any pending events.

# Index

---

## A

assembler, 16, 24  
  CPU window, 74  
  stack frames, 44  
associations  
  OS/2, 28  
  Windows, 28  
auto indent, 38  
auto save, 37

## B

backup files, 38  
blocks  
  persistent, 38  
breakpoints, 64. *See also* debugging  
  advanced options, 70  
  data, 70  
  execution, 70  
  threads and, 70

## C

call stack, 70  
calling conventions  
  assembler code and, 24  
  C, 26, 45  
  pas16, 45  
  pascal, 45  
  stdcall, 26, 45  
command line, 69  
command line tools, 51  
comments  
  nested, 48  
compatibility. *See also* features  
  Borland Pascal, 48  
  classes, 23  
  Pointers, 23  
compatibility, 21  
Compatibility  
  Reals, 23  
compile  
  build, 34  
  compile, 34  
  make, 34, 41  
  target platform, 41  
compiler  
  command line, 51  
  configure, 43  
compiler directives, 17  
  **\$B**, 46

**\$D**, 48, 74  
  **\$H**, 47  
  **\$I**, 46, 52, 71  
  **\$J**, 48  
  **\$L**, 48  
  **\$M**, 49  
  **\$Q**, 46  
  **\$R**, 36, 46, 48, 52  
  **\$S**, 46  
  **\$T**, 47  
  **\$V**, 46  
  **\$W**, 44  
  **\$X**, 47  
  **\$Z**, 45  
  **\$Zd**, 48  
  **&AlignCode**, 44  
  **&AlignData**, 44  
  **&AlignRec**, 44  
  **&Asm**, 48  
  **&CDecl**, 45  
  **&Comments**, 33, 48  
  **&Delphi**, 47  
  **&Delphi-**, 61  
  **&far16**, 45  
  **&Frame**, 44  
  **&G3**, 45  
  **&G4**, 45  
  **&G5**, 45  
  **&Linker**, 19, 51  
  **&LocInfo**, 50, 52, 63  
  **&Open32**, 26, 48  
  **&Optimize**, 26, 45, 74  
  **&Optimize**, 45  
  **&OrgName**, 44  
  **&PMTType**, 49  
  **&Size**, 45  
  **&SmartLink**, 46, 48  
  **&StdCall**, 45, 48  
  **&Use32**, 21, 48, 60  
  automatically insert, 61  
  ignored, 24  
  VP specific, 17  
  VPC and, 52  
Compiler directives, 24  
compiler. *See also* VPC  
compression, 18  
conditional compilation, 48  
controls, 28  
conventions, 7  
CPU window, 73  
CUA setup, 39

---

## D

debugger

- API, 82
- break on exception, 62
- debug startup code, 62
- display swapping, 61
- extending, 82
- hard PM mode, 62
- integer format, 61, 67
- local menu, 66
- origin, 65
- run in window, 62
- user screen delay, 62
- debugging, 58
  - advanced, 70
  - animate, 69
  - breakpoints, 64
  - call stack, 70
  - coprocessor, 73
  - CPU window, 73
    - CPU pane, 74
    - example, 74
    - local menu, 75
  - dump window, 72
  - evaluate expression, 66
  - fill memory block, 73
  - go to cursor, 69
  - inspect, 66
  - instruction trace, 69
  - modify variable, 66
  - object hierarchy, 72
  - registers, 73
  - reset, 63
  - restart, 63
  - run program, 62
  - run-time errors, 63
  - semantic errors, 58
  - show last error, 59, 63
  - single step, 69
  - stand-alone executables, 76
  - step over, 69
  - threads, 71
  - trace into, 69
  - unit list, 71
  - watch expressions, 66
  - why, 58
- description, 6
- direct port I/O, 17
- directories, 14
  - by platform, 14
  - exe output, 50
  - library, 50
- directory structure, 11
- DOS extenders, 82
- DOS stub, 18
- drive substitution, 37, 83, 84
- dump window, 72

---

## *E*

editor

- CUA, 39
  - syntax highlight, 38, 39
- e-mail address, 8
- environment variables
  - VPHELP, 33
- evaluator, 66
- example
  - first program, 41
  - second, 42
- examples
  - Borland Pascal, 78
- exceptions
  - catching, 65
  - CPU, 65
  - language, 62
- expanding VP, 81
  - debugger support, 82
  - linker support, 81
  - RTL support, 81

---

## *F*

- features, 13
  - Borland Pascal, 16
  - Delphi 1, 16
  - Delphi 2 and 3, 16
- files
  - .ASM, 48
  - .CFG, 40, 54
  - .DEF, 19, 50
    - creating, 54
  - .DLL, 17, 49, 50
  - .EXE, 16, 17, 49
  - .HLP, 55
  - .LIB, 36, 46, 49, 50, 52, 54, 81
  - .MAP, 16
  - .OBJ, 16, 36, 46, 49, 52, 81
  - .PAS, 52
  - .RES, 18
  - .SYM, 18, 27, 76
  - .TPH, 56
  - .VPH, 55, 56
  - .VPI, 16, 36, 52
  - .VPO, 27, 39, 52
  - executable, 36
  - import libraries, 36
  - import library, 50
  - include, 36
  - locations, 11, 36
  - map, 18
  - resource files, 37
  - units, 36

---

## *H*

- Help converter, 33, 56
- Help generator, 55

history, 7

---

### I

IDE, 13

- basics, 28
- blocks, 31
- colours, 29
- configuring, 35
- editing, 30
- help, 33
- local menus, 29
- macros, 32
- menu bar, 33
- navigating, 30
- redo, 32
- searching, 31
- undo, 32
- VPPM for OS/2, 13
- windows, 29

ImpLib, 54

inline, 24

insert mode, 38

inspector window, 66

- settings, 67

- zoom, 67

installation, 10

- source code upgrades, 11

---

### K

Keyboard

- Windows, 84

---

### L

linker. *See* options:linker

- external, 50

- internal, 49

Linux, 12, 81

location information, 19, 22, 50, 63

log window, 70, 71

Lst, 22

---

### M

macros

- recording, 32

- restrictions, 33

- stop recording, 33

map files, 18

Mem array, 23

menus

- compile, 34

- debug, 34

---

- edit, 34

- file, 34

- help, 35

- local, 29

- options, 34

- run, 34

- search, 34

- view, 34

- window, 34

---

### N

NCP window, 73

nested comments, 48

---

### O

object hierarchy, 72

Object Windows Library, 25

ODAPI. *See* debugger

Open32, 19, 26

options

- auto save, 37

- calling convention, 45

- changing directory, 37

- code generation, 44

- colours, 39

- compiler, 43

- defaults, 43

- conditional defines, 48

- CPU, 45

- 80386, 45

- i486, 45

- Pentium, 45

- debugger, 61

- debugging, 48

- directories, 36

- editor, 37

- error checking, 46

- linker, 43, 49

- application, 49

- compression, 49

- map file, 50

- messages window, 37

- mouse, 39

- optimisation

- debugging, 45

- options, 45

- speed vs size, 45

- resource compiler, 36

- save, 39

- screen size, 37

- source tracking, 37

- stack size, 49

- starting directory, 37

- syntax, 46

---



unit aliases, 49

---

## P

parameters, 69  
 program type  
   VIO, 41  
 project  
   configuration file, 40

---

## R

record  
   **packed**, 44  
 registers window, 73  
 resource files, 18  
 RTL, 17  
   Crt, 22  
   Dos, 22  
   Printers, 22  
   System, 21, 22  
   SysUtils, 19, 21  
   Use16, 21  
   Use32, 21  
   VpSysLow, 17, 21  
   Windows, 18  
 RTLI. *See* location information  
 RTTI, 16, 23  
 RTXDOS, 82  
 Run-Time Library. *See* RTL

---

## S

semicolons, 59  
 smart linking, 17  
 smart tabs, 38  
 stub program, 18  
 support, 8  
 syntax highlight, 33, 38  
   file extensions, 39  
 sysplatformname, 84  
 system requirements, 7

---

## T

target platform, 34, 41  
   command line compiler, 84  
   restrictions, 84  
   user-defined, 82  
 threads. *See* debugging  
 Turbo Vision, 25  
 types  
   AnsiString, 47  
   integers, 21, 22, 60  
   pointers, 23  
   reals, 23  
   ShortString, 48  
   string, 47

---

## U

unit  
   introducing, 42  
 UNIX text, 38  
 uses clause, 42

---

## V

variables  
   initialised, 48  
   storage, 66  
 Visual Component Library, 26  
 VPC  
   compiler directives, 52  
   options, 51  
   response file, 54  
   settings file, 51  
   syntax, 51

---

## W

watch window, 66  
   local menu, 66  
 window  
   messages, 42