Visual Programming Interface

(c) Copyright 1984-2001
Prolog Development Center A/S
H.J. Holst Vej 3-5C,
DK-2605 Broendby, Copenhagen
Denmark
Copyright

The documentation for this software is copyrighted, and all rights are reserved. It may not be reproduced, transmitted, stored in a retrieval system, or translated, either by electronic, mechanical or any other means, without the prior written consent of Prolog Development Center A/S.

The software products described in these manuals are also copyrighted, and are licensed to the End User only for use in accordance with the End User License Agreement, which is printed on the disk packaging. The prospective user should read this agreement carefully prior to use of the software.

Visual Prolog is a registered trademark of Prolog Development Center A/S.

Other brand and product names are trademarks or registered trademarks of their respective holders.
Table of Contents

INTRODUCTION ............................................................................................... 1
   The Visual Programming Interface.............................................................. 1
   Event-driven Applications............................................................................. 2
   About this Book.......................................................................................... 3

Part 1 The VPI Layer

Chapter 1, Invoking the VPI

Chapter 2, Windows
   Window Types.............................................................................................. 8
   Window Styles............................................................................................. 9
   The Client Rectangle ..................................................................................... 10
   The Screen Window ...................................................................................... 11
   The Task Window ......................................................................................... 11
   Top-Level Windows ..................................................................................... 11
   Child Windows ........................................................................................... 12
   Controls....................................................................................................... 12
   Dialogs........................................................................................................ 12
   Event-Handlers ........................................................................................... 13
      Event Handler Return Values ................................................................... 13
   Creating Windows ....................................................................................... 14
      Creating a Regular Window ...................................................................... 14
      Creating a Dialog..................................................................................... 15
      Creating a Control................................................................................... 15
      Creating Windows Dynamically............................................................. 15
   Destroying Windows .................................................................................... 17
   Retrieving the Client Rectangle .................................................................. 17
   The Clipping Rectangle ............................................................................... 17
   Retrieving the Outer Rectangle ................................................................... 18
   Calculating Client Rectangle from Outer Rectangle ............................... 18
   Moving and Re-sizing Windows .................................................................. 18
   Changing the State for a Window............................................................... 19
      Disabling and Enabling Windows ............................................................ 19
      Hiding and Showing Windows ................................................................. 20
      Minimizing, Maximizing and Restoring Windows ................................... 20
   Changing the Icon Associated with a Window........................................... 20
   Changing the Window Text or Window Title.............................................. 20
   Changing the Event-handler ....................................................................... 21
Connecting Data to a Window ........................................................................................................... 21
Finding the Task Window ....................................................................................................................... 21
Finding the Parent Window ..................................................................................................................... 21
Finding the Active Window ..................................................................................................................... 22
Setting the Focus ........................................................................................................................................ 22
Finding the Window which has the Focus ................................................................................................. 22
Window Ordering ...................................................................................................................................... 22
Window Updating ..................................................................................................................................... 22
  Invalidate Window ................................................................................................................................. 23
  Validate Part of the Window ................................................................................................................... 23
  Check whether Part of the Window Needs Updating .............................................................................. 23
  Force Updating of the Window .............................................................................................................. 24

Chapter 3, Events
  Window Creation ...................................................................................................................................... 25
  Window Being Destroyed .......................................................................................................................... 26
  User Requests to Close Window ............................................................................................................ 26
  GUI is Closing Down ............................................................................................................................... 26
  Menu Becomes Active .............................................................................................................................. 27
  A Menu Selection has been Made ......................................................................................................... 27
  Mouse Click ........................................................................................................................................... 28
  Mouse Double Click ............................................................................................................................... 29
  Mouse Moved in Window ....................................................................................................................... 29
  Mouse Button Released ......................................................................................................................... 29
  Keyboard Input ....................................................................................................................................... 29
    A Key has been Pressed Down ............................................................................................................ 30
    A Key has been Released .................................................................................................................... 30
    Horizontal Scroll Event .................................................................................................................... 30
    Vertical Scroll Event ........................................................................................................................... 30
    Focus is being Gained ......................................................................................................................... 31
    Focus is being Lost ............................................................................................................................... 31
    Background Erasing ............................................................................................................................ 31
    Window Location has Changed ........................................................................................................... 31
    Window has Changed Size .................................................................................................................. 32
    Window has Changed State .................................................................................................................. 32
    Native GUI Events ............................................................................................................................... 33
    Owner Drawing ................................................................................................................................... 33
    Return Owner Draw Size ..................................................................................................................... 33
    Updating the Window ............................................................................................................................ 33
    Timer Interval has Expired .................................................................................................................... 34
    A Programmed Event has Occurred .................................................................................................... 34
    Dynamic Data Exchange ...................................................................................................................... 34
Control Notification Events............................................................................. 35
The CONTROL_INFO Domain .................................................................. 35
   Events for Scroll bars ...................................................................... 35
The Control is Gaining focus ............................................................. 36
The Control is Losing focus ............................................................... 36
Control Content has been Modified .................................................. 36
Events for List boxes.......................................................................... 36
Control is activated by mouse or keyboard ........................................ 36
Application Termination Notification Events ................................................. 37
Application has Activated................................................................................ 37
Application has been Deactivated ...................................................... 37

Chapter 4, Controls
Manipulating Controls .............................................................................. 39
Style Flags........................................................................................................ 39
Events from Controls............................................................................... 47
Creating Controls........................................................................................... 48
The Various Controls.................................................................................. 49
   Group Box Controls............................................................................. 49
   Icon Controls.......................................................................................... 49
   Static Text Controls............................................................................... 50
   Edit Controls........................................................................................... 50
      Edit controls send three notification events: .................................. 50
   Push Buttons Controls .......................................................................... 50
      Default Push Button ..................................................................... 51
      Notification Events for Push Buttons ........................................... 51
   Check Box Controls............................................................................... 51
      Check Provide Notification of Only One Type of Event: ............... 52
   Radio Button Controls ........................................................................... 52
      Notification Events from Radio Buttons ....................................... 53
   List Box Controls.................................................................................. 53
      List Box Predicates ....................................................................... 53
      Notification Events from List Boxes ............................................. 57
   List Buttons............................................................................................ 58
      Notification Events from List Buttons ......................................... 58
   List Edit Boxes...................................................................................... 59
      Notification Events from List Edit Boxes ....................................... 59
   Scroll Bars .............................................................................................. 59
      Notification Events from Scroll Bars ............................................ 59
   Scroll Bar Referencing ......................................................................... 61
   Scroll Bar Range.................................................................................... 61
   Scroll Bar Positioning .......................................................................... 61
Chapter 5, Dialogs
Common Dialogs ............................................................................................. 67
Get User Response to Yes/No etc. Question ............................................... 67
User Selects a Filename(s) to Opening/Saving a File(s) ............................. 68
User Input of a String .................................................................................. 69
User Selects from a List of Strings .............................................................. 69
User Selection and Set up of a Printer ........................................................ 70
User Selection of a Color ............................................................................. 71
User Selection of a Font .............................................................................. 72
Show an Error ............................................................................................. 72
Show a Note ................................................................................................. 73
User Defined Dialogs ...................................................................................... 73
Control Identifiers ........................................................................................ 73
Special Control Resource identifiers .................................................. 74
Creating a Dialog ........................................................................................ 74
Initializing a Dialog ..................................................................................... 75

Chapter 6, Drawing Operations
Drawing Tools ................................................................................................. 76
Pens .............................................................................................................. 77
Get the Current Pen ..................................................................................... 77
Setting the Pen to be Current for a Window ............................................. 77
Background Modes ..................................................................................... 78
Brushes ......................................................................................................... 78
Getting and Setting the Current Brush ..................................................... 79
Fonts ............................................................................................................. 79
Drawing Modes ............................................................................................ 79
Getting and Setting the Current Drawing Mode ...................................... 79
Drawing Predicates ..................................................................................... 80
Drawing a Pixel .......................................................................................... 81
Filling an Area .............................................................................................. 81
Drawing Icons ............................................................................................. 81
Drawing Open Figures ............................................................................... 81
Drawing Filled Figures ............................................................................... 81
Drawing Text ............................................................................................... 82
Measuring Text ............................................................................................ 83
Coloring Text ............................................................................................... 83
Font Handling .............................................................................................. 84
Font Creation ............................................................................................... 84
# Table of Contents

- **Font Selection** ......................................................... 85
- **Obtaining Font Information** ......................................... 85

## Chapter 7, Color Handling

## Chapter 8, Coordinate Systems
- Dialog Base Units .......................................................... 91
- Mapping, Scaling, Panning and Zooming .......................... 92
  - Mapping Examples ......................................................... 94
    - Drawing a Map ............................................................ 94
    - Printing ................................................................. 95

## Chapter 9, Printing
- Start a Print Job ............................................................. 97
- End a Print Job ............................................................... 97
- Set a Page Orientation .................................................. 97
- Start a New Page .......................................................... 98
- End a Printed Page ....................................................... 98
- Abort a Print Job .......................................................... 98
- Retrieve Printing Settings .............................................. 98

## Chapter 10, Printing
- Cursor Predicates .......................................................... 101
  - Examples ................................................................. 102

## Chapter 11, Mouse Handling
- Capturing the Mouse ..................................................... 103
- Mouse Events .............................................................. 103

## Chapter 12, Rectangles
- Intersection of Two Rectangles ....................................... 105
- Inflate the Size of a Rectangle ......................................... 105
- Determine Whether the Area of a Rectangle is Zero ............ 105
- Move a Rectangle ........................................................ 106
- Determine Whether a Point is Inside a Rectangle ............... 106
- Create the Union of Two Rectangles ............................... 106
- Process Pending Events ............................................... 107

## Chapter 13, Event Handling
- Sending and Posting Events .......................................... 108

## Chapter 14, Resources
- Resource Naming .......................................................... 110
- Resource Linking ........................................................ 111
- Bitmaps, Icons and Cursors ......................................... 111
- Dialog Resources ......................................................... 111
- Menu Resources .......................................................... 111
Changing the Values in a Toolbar ................................................................. 175
Creating Radio Buttons in a Toolbar.............................................................. 175

Chapter 32, Tree Package
The Domains in the Tree Tool................................................................. 176
The Predicates in the Tree Tool.............................................................. 177
Creating a Window with a Tree.............................................................. 177
Creating a Tree in an Already Open Window (Subclass) ....................... 177
Return a List of All Nodes......................................................................... 177
Return the Current Selected Node .......................................................... 177
Return the Parent Node for the Current Selected ..................................... 177
Return the Node Type for the Current Selected ........................................ 177
Return Settings from the Tree Window.................................................... 177
Return Rectangle for the Tree Window.................................................... 177
Return Current Search Value - or Fail..................................................... 177
Move Selection to the First Node Matching the Selector.......................... 177
Sets New Direction for the Tree ............................................................... 178
Sets new Font for the Tree................................................................. 178
Set a New Tree in the Window............................................................... 178
Explode/Implode Subtree from the Selected Node.................................. 178
Select First Node Using the Selector....................................................... 178
Select next node using the chosen Selector.......................................... 178
Make Selected Node Visible ............................................................... 178
Move Selection from Child Node to the Parent Node............................. 178
Print the Tree...................................................................................... 178

Chapter 33, Ownerdrawing Package
Index........................................................................................................ 181
INTRODUCTION

The Visual Programming Interface

The Visual Prolog (VP) product consists of several elements. It has an interactive Visual Development Environment (VDE) that includes text and various graphical editors, code generating tools (Experts), build control logic, and extensions to Prolog in the form of a Visual Programming Interface (VPI). It also includes a Prolog compiler, various include files and libraries, a linker, and various example and help files.

Visual Prolog is designed to make it easy, convenient and fast to develop, test, and modify applications written in PDC Prolog. It supports many platforms, including MS-DOS, MS Windows 3.x, MS Windows 95/98/ME, Windows NT/2000, as well as 16 bit and 32 bit OS/2 target platforms using processors from the 80x86 base family.

In the Application Expert, for User Interface Strategy you should select the Visual Programming Interface (VPI). This provides access to a set of Code Experts, resource editors, and additional VPI-specific predicates, definitions, and libraries. The resource editors are used to define, layout, and edit windows, dialogs, menus, tool bars, help bars, string tables, icons, cursors, bitmaps, and run time on-line help. From these, Code Experts generate the necessary Prolog source code framework to present these components in some presentation space (i.e. in a window). This framework is ready to be compiled, linked, and run.

At the programmer’s request, the Code Experts can bring up any part of its generated framework in an editor window for inspection and filling out with application logic to transform the framework into a useful application. This completion process in an editor window is greatly assisted by a variety of commonly needed functions being available at a click of the mouse right button from a multilevel pop up menu of editing, selecting, searching, replacing, and inserting functions. Much code can be accurately generated by use of the insert item of this menu which permits selection of colors, VPI and standard Prolog predicate templates, user-defined predicates for dialogs and windows, domain names, resource identifiers, VPI constants, quoted full path and file names as obtained by directory browsing, file names, key words, and much more.

The Visual Programming Interface (VPI) is a high-level Application Programming Interface (API) and is designed to make it easy for Prolog
applications to provide sophisticated user interfaces utilizing the graphical capabilities of today's operating systems and display hardware. The Resources and Tools needed by such applications - Windows, Menus, Dialogs, Controls, Pens, Brushes, Cursors, Carets, Pictures, etc. all appear as simple Prolog structures.

The VPI makes it possible to create portable source code that can be cross-compiled to run in 16-bit mode under MS-Windows, 32-bit mode under Windows 95/98/ME, Windows NT/2000, under the OS/2 presentation manager.

The VPI is an abstraction of the supported Graphical User Interface (GUI) platform APIs so the functions, flags and events are handled independently of the platform. However, not all facilities of every GUI appear in the VPI, nor is every facility in the VPI fully portable; rather, the VPI tries to support all the features found on each platform in a uniform way.

The domains, constants and predicates in the VPI do not change with target platform. To assist developers not interested in portability, but who need maximum flexibility on one platform, the VPI also contains a number of predicates that are not portable, but are very relevant on a given platform. Just as when using other general purpose programming languages, it is possible to write a portable VPI application, but it is also possible to write one that is not portable.

**Event-driven Applications**

In most DOS text mode applications the sequence of operations is determined by the programmer at implementation time. There is very little interaction with other applications and no interference from the operating system. Most things are completely controlled by the programmer. A DOS application communicates directly with its user and is often based on a static design philosophy.

Modern GUI systems are much more dynamic in their nature. The operating system doesn’t sit back and wait any more. Rather than only providing a layer of services, it is actively involved in the execution sequencing of all applications. An application talks to its user indirectly via the operating system. When the user does something (presses a key, moves the mouse, clicks on an icon etc.) an event is generated by the hardware interface, the operating system translates this event into a standardized message format and sends the event notification message to its correct owner, usually the window that is active (said to 'have the focus')

These events can arrive at the application asynchronously, and so the application must be written most carefully in order to maintain the integrity of all its data.
and processes no matter what the sequence of events. GUI-based applications must be reactive to the world around them and cannot mimic the proactivity and imperial behavior of DOS applications. The applications behind inactive windows remain active and running. They may write or draw into their inactive windows at any time and this will be quite visible unless the window is covered or overlapped by another. An application may at any time suddenly activate one of its own windows to request input. The previously active window becomes inactive, but its application continues running.

In the middle of entering a string into an edit box, the user may receive an urgent telephone call. The user may respond by selecting your Task window’s system button or file menu and requesting the shut down of the application. The operating system will inform the application of the event by sending a message to the Task window's event-handler, and the programmer should have arranged to do in an orderly fashion any necessary house-keeping, then terminate the application by destroying the Task window. In many multi-tasking systems the operating system can suspend your application at any time. The application will never know if the user requested a shutdown or if, for example, the operating system has detected a coming power failure. It just receives a message and acts accordingly. This flexibility is not confined to the operating system and to your application’s user. If the application wants to close some window it can send that window an e_CloseRequest, just like the one it receives when your user selects 'Close' from that Task window menu.

Of course you can reduce the quantity of possible events and messages by disabling menu items and push buttons that are not relevant to the task at hand. The message handling part of GUI applications can be very complex, and this is the price for the great flexibility these systems give you and your users. In a C application the result is often an enormous and unwieldy case statement. Prolog and the VPI make life a lot easier for you than if you interface the application to the native GUI directly. The Code Expert and the Browser make it easy to find and maintain the code that handles a specific event by responding to a specific message and you rarely have to deal with the intricate details of the underlying messaging logic.

**About this Book**

Before reading this book, you should have knowledge about the Prolog language. You can get this knowledge by reading the Language Tutorial manual, and working your way through the examples in that book.
Also, before starting on this book, you should study the Guided Tour in Visual Prolog in the getting started manual. The Visual Development Environment of Visual Prolog is tightly coupled to the VPI layer, and can give much assistance in using this. The high productivity of Visual Prolog comes from a proper knowledge about using the Layout Editors and Code Experts in conjunction with the VPI and the Prolog language. The Guided Tour gives you an overview of coupling these things together.

This book gives a detailed introduction to the VPI layer. It explains the concept of Windows, the various attributes and functions calls. After this, it describes the Event handling, and working with controls and dialogs. After these basic things are introduces the various groups of Windowing operations are introduces one at a time. The book ends by describing some tool packages, that are implemented on top of the basic VPI layer.

This manual are part of the on-line help of Visual Prolog, but the on-line help does also contain one entry for each VPI predicate, giving the full description about each predicate.

The functionality of the VPI is illustrated by various examples in the subdirectory VPI\EXAMPLES.

For full and accurate declarations of all the VPI layer components, the reader is referred to the content of the three files VPI.CON, VPI.DOM and VPI.PRE in the VPI\INCLUDE directory.
PART 1

The VPI Layer
Invoking the VPI

Invoking the VPI Execution of a VPI application starts with the usual Prolog GOAL. The basic function of the GOAL is to invoke the predicate \textit{vpi\_Init}, which will start the VPI windowing system, and create one main window, which we call the Task window. The only code which should precede the call to \textit{vpi\_Init} is that involved with setting the parameters of \textit{vpi\_Init}. These can all be set by the predicate \textit{vpi\_SetAttrVal}.

The Application Expert will create the GOAL section for you, with all applicable attributes and flags set according to the choices you made in the Application Expert dialog. Any other application initialization operations should be done in response to the \textit{e\_Create} event for the Task window. All termination should be done in response to the \textit{e\_Destroy} event.

\begin{verbatim}
GOAL
    ....
    vpi_Init(WSFlags, EventHandler, Menu, AppName, TaskWinTitle).
\end{verbatim}

Notice that \textit{vpi\_Init} will return only after the user closes the created Task window.

\textit{WSFlags} is a list of the window style flags for the Task window. These determine whether it should start minimized, maximized, etc. \textit{Menu} is from the MENU domain \texttt{(res\_menu (resource\_id) or no\_menu)}. \textit{EventHandler} is the predicate from the EHANDLER domain that will receive events for the Task window. \textit{AppName} is used to identify the application. Finally, \textit{TaskWinTitle} will become the title of the Task window. These flags are described in more detail later under Creating Windows.
Windows

A window is a rectangular presentation space on a display. A window is used by the computer application to present output in and receive input from the user. The entire screen space is treated as a special window called "the Screen window". Every window shares the screen with other windows, including those from other applications and sometimes from the operating system itself, and these windows may overlap.

No matter how many applications are concurrently running, only one window at a time is active in the sense that it can receive input from the user. An active window is displayed and is in full color, so that, using a desktop metaphor, it appears to be "on top". The user can use the mouse, the keyboard, or other input devices to communicate with this window - and thereby with the underlying application that owns it. Windows are the only immediate data turn-around device that an application has to interact with its current user. Therefore, one of the first functions of every application is to create and show a window. This first-created window for each application is called its Task window, and while it need not necessarily be shown, it does represent the application as a whole, as a receptor of messages.

All of the VPI run-time window creation predicates create a memory structure, display the window according to this control structure, and return a window handle. A window handle is a unique identifying number that the VPI assigns when a given window is created. Not only visible windows have handles, but also non-displayed or not-yet-displayed windows used for drawing or printing. Nearly all the other VPI predicates require a window handle as input in order to indicate which window to access. Loaded and ready-for-use resources of all kinds are also issued handles by which they are accessed. The actual value of any handle is irrelevant to the application and should be ignored.

Every window or dialog in an application is represented in the application not only by a memory structure, but also by an event-handling predicate, to which the VPI sends messages when user or system events occur. These messages can arrive at any time and need not be synchronized with the application. The application can also send event messages to any of its event-handlers. In the VPI, names in the event message domain always have an 'e_' prefix.
Window Types

Windows are classified by type. Window types must belong to the WINTYPE domain, and have 'w_' (for windows), 'wc_' (for controls), or 'wd_' (for dialogs) prefix. The window type can be obtained from the window handle by a call to the \texttt{win\_GetType} predicate. The possible window types are listed in the table below.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Identifier & Meaning                   \\
\hline
\hline
\texttt{w\_TopLevel} & Normal document window       \\
\texttt{w\_Child}    & Child window                 \\
\texttt{w\_Task}     & Task window                  \\
\texttt{w\_Screen}   & The display Screen Itself    \\
\texttt{wd\_Modal}   & Modal dialog                 \\
\texttt{wd\_ModeLess} & Modeless dialog              \\
\texttt{wc\_PushButton} & Push button control         \\
\texttt{wc\_RadioButton} & Radio button control        \\
\texttt{wc\_CheckBox} & Check box control           \\
\texttt{wc\_HScroll} & Horizontal scroll bar control \\
\texttt{wc\_VScroll} & Vertical scroll bar control \\
\texttt{wc\_Edit}    & Edit control                 \\
\texttt{wc\_Text}    & Static text control         \\
\texttt{wc\_LBox}    & List box control            \\
\texttt{wc\_LBoxButton} & Button with drop-down list  \\
\texttt{wc\_LBoxEdit} & Edit field with drop-down list \\
\texttt{wc\_GroupBox} & Group box                    \\
\texttt{wc\_Icon}    & Icon control                 \\
\texttt{wc\_Custom}  & Custom control              \\
\hline
\end{tabular}
\caption{Window Types and Classes}
\end{table}
Window Styles

When you create a window or a dialog there are various flags to determine how the window should look, what kind of border it should have, whether it should have scroll bars, whether it should be possible to maximize it, etc. You use the VDE's Window and Dialog editor to do this. All flags have a 'wsf_' prefix. After you create a window, style flags for the window may be obtained by a call to `win_GetState`. Some may be changed by a call to `win_SetState`. A table of all the possible values follows:

<table>
<thead>
<tr>
<th>Style Flag</th>
<th>Meaning that ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsf_Border</td>
<td>Window gets thin border.</td>
</tr>
<tr>
<td>wsf_SizeBorder</td>
<td>Window gets double border. Used to re-size the window.</td>
</tr>
<tr>
<td>wsf_DlgBorder</td>
<td>Window gets border like a dialog. Window cannot be re-sized.</td>
</tr>
<tr>
<td>wsf_Close</td>
<td>Window gets System Menu. Generally used to close Window.</td>
</tr>
<tr>
<td>wsf_Maximize</td>
<td>Window gets maximize button. Do not specify this flag if the window should not be re-sized.</td>
</tr>
<tr>
<td>wsf_Minimize</td>
<td>Window gets minimize button.</td>
</tr>
<tr>
<td>wsf_HScroll</td>
<td>Window gets horizontal scroll bar.</td>
</tr>
<tr>
<td>wsf_VScroll</td>
<td>Window gets vertical scroll bar.</td>
</tr>
<tr>
<td>wsf_TitleBar</td>
<td>Window gets title bar.</td>
</tr>
<tr>
<td>wsf_Visible</td>
<td>Window is initially visible. This is the default.</td>
</tr>
<tr>
<td>wsf_Invisible</td>
<td>Window should be created invisible.</td>
</tr>
<tr>
<td>Flag</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>wsf_Enabled</code></td>
<td>Window should be created initially enabled. This is default.</td>
</tr>
<tr>
<td><code>wsf_Disabled</code></td>
<td>Window should be created initially disabled</td>
</tr>
<tr>
<td><code>wsf_Maximized</code></td>
<td>Window should be created initially maximized</td>
</tr>
<tr>
<td><code>wsf_Minimized</code></td>
<td>Window should be created initially minimized</td>
</tr>
<tr>
<td><code>wsf_Restored</code></td>
<td>This flag is used to restore a window from the minimized or maximized state.</td>
</tr>
<tr>
<td><code>wsf_ClipChildren</code></td>
<td>Drawing over any Child windows should be clipped. This flag must be given when creating the parent window.</td>
</tr>
<tr>
<td><code>wsf_ClipSiblings</code></td>
<td>Clipping should be done when Child windows overlap. This flag can be used only for Child windows.</td>
</tr>
<tr>
<td><code>wsf_NoClipSiblings</code></td>
<td>Deactivates the <code>wsf_ClipSiblings</code> flag.</td>
</tr>
<tr>
<td><code>wsf_Topmost</code></td>
<td>Window will be on top of all other non-TopMost windows.</td>
</tr>
<tr>
<td><code>wsf_Transparent</code></td>
<td>Window should be transparent, and not damage any windows underneath it.</td>
</tr>
</tbody>
</table>

### The Client Rectangle

The portion of any window inside any title, menu, and scroll bars is termed the Client Rectangle. The Client Rectangle is the working area of a window - it is available for controls, child windows, coloring, and the drawing of text and graphics.

Rectangles are always specified as RCT structures: `rct(Left, Top, Right, Bottom)`.

Note that the VPI uses `Right, Bottom NOT Width, Height`.

For most windows coordinates are expressed in pixels. This applies when working in windows on the screen with editors, menus, trees, toolbars, etc.

But when working with dialogs and controls, sizes should be dependent on the font size and type used of the system font. When the system font changes, the size of text controls will change and the whole dialog will be scaled accordingly. Therefore, the size of dialogs is measured in Dialog Base Units, whose metrics depends upon the system font.
The Screen Window

The Screen window is an abstraction that represents the entire screen. The Screen window is always the parent of the Task window. Top-Level windows may also have the Screen window as their parent, which allows moving them outside of the Task window.

The Task Window

The Task window is an abstraction, which represents the application itself. On Windows and OS/2 it can be used much like a normal window, but on the Macintosh and under Motif this may not be so - it may not display on some platforms. The Task window is the main window for an application and is created with the \texttt{w\_Task} window type by a call to the VPI predicate \texttt{vpi\_Init}. This is generated by the Code Expert as the final call in the application goal.

Application initialization and termination are always associated with the Task window. Initialization should be programmed in response to the \texttt{e\_Create} Task window event, which is triggered automatically by a call to the \texttt{vpi\_Init} predicate. The clean-up on termination should be programmed in response to the \texttt{e\_Destroy} event. Application termination results from the VPI predicate \texttt{win\_Destroy} are applied to the Task window. One way this might occur would be in response to the user selecting the \texttt{File} \texttt{| Exit} menu item. This will typically trigger an event resulting in a call to the \texttt{win\_Destroy} predicate.

The Task window also receives some special events that are not sent to other windows. An \texttt{e\_EndSession} may be received from the windowing system to indicate that the windowing system is being shut down, and the various Dynamic Data Exchange (DDE) events used to communicate with other applications are also sent to the Task window's event-handler.

Under Windows' Multiple Document Interface (MDI) mode, the Task window is the container for all Top-Level windows created by the application and for their minimized icons, and it is the Task window that carries the menu of any active MDI window, otherwise it carries its own. When running in MDI mode, it is not possible to draw in the Task window.

Top-Level Windows

In the VPI, a Top-Level window is a window that has either the Screen window or the Task window as the parent window. Top-level windows are created with the \texttt{w\_TopLevel} window type. These windows will often contain a menu, and
they can usually be minimized and maximized. They may also be used as container windows, and may carry Child windows or controls.

In MDI mode under Windows, the menus of Top-Level windows will be displayed in the Task window. However, in SDI (single document interface) mode, any Top-Level window can nominate the Screen window as its parent window. This allows each Top-Level window in the application to contain its own menu.

**Child Windows**

Child windows are always restricted to stay inside their parent window. Child windows are created by requesting the `w_Child` window type. When a parent window is destroyed, minimized, disabled, or hidden, the operation is automatically performed for all of the Child windows and controls before being done for the parent.

**Controls**

The term Controls applies to smaller windows with a specialized appearance and fixed functionality supplied by the operating system. They are used in fixed arrangements inside other windows, mainly as components of dialog boxes, but they can also be used in other windows. Controls provide the user with the means to type text, choose options, or invoke commands. Commonly available named types of controls are supported by the VPI and these names commence with a `wc_` prefix. Their events cause notification messages to be sent to their parent window event-handler, which would generally be the dialog event-handler.

**Dialogs**

A Dialog box is a special kind of window, which will be referred to throughout this user guide simply as a "dialog". They are restricted in functionality compared to regular windows. Dialogs are normally filled with controls, which can interact with the user by displaying output, accepting input, offering selection from a number of options, or allowing editing. The operating system underlying the VPI provides the ability to navigate between controls by using the Tab and Shift+Tab keys to shift the point of user interaction from one control forwards and backwards to another.
Event-Handlers

Each window and dialog has an associated predicate, which handles all events for the window. It should be declared as belonging to the EHANDLER domain and takes two input arguments: a handle to the window and the event. The predicate has a LONG return final parameter, whose value is only used in a few cases. A sample event-handler might look like:

PREDICATES
  mywin_event_handler : EHANDLER

CLAUSES
  mywin_event_handler(Win, e_Create(CreationData), 0):-

The event-handler predicate for each window should be declared and created by the Dialog and Window Expert. You should generate event code only for the events that need to be handled explicitly by your application. If the event-handler predicate for that event fails, the VPI will perform default processing appropriate to the event. If the predicate succeeds, the VPI will perform no extra processing. As an example, the e_CloseRequest event is sent when the user double clicks in the close box for a window. If there is no clause to catch an e_CloseRequest event the default handling will be to close the window. If there is a clause to catch an e_CloseRequest event, the programmer can decide to leave the window open by having the predicate succeed, or close it by either failing or by doing a win_Destroy.

If no event-handler clause for an event exists or the event-handler clause for that event fails the VPI will perform default processing appropriate to the event. If the event-handler clause succeeds no additional or default handling is performed.

Event Handler Return Values

If the return parameter of an event-handler predicate does not carry any special meaning it should be set to zero. Generally, this is the case for most events. There are some cases where it has special meaning. These include: e_Native events to give a return value to the underlying GUI; to give a return value in response to an e_User event arising from a call to win_SendEvent; to return a size in response to e_OwnerMeasureItem; or to return data to another application in a DDE conversation.
Creating Windows

In Visual Prolog applications, the creation a window is normally done in the code generated by the Dialog and Window Expert. The Code Expert will here use all the settings specified in the Attribute dialogs and the window layout to create the windows.

Creation of a window, a dialog or a control is performed by one of the VPI predicates starting with \texttt{win\_Create}. They all return a handle to the newly created window. Note, that if the window or dialog created was modal, the window will already have been closed when we return, so the returned window handle will be meaningless in those cases.

When a window is created, the user must specify the desired client rectangle for the window. The coordinates are Top, Left, Right, and Bottom relative to the client rectangle of the parent window.

None of the VPI window and dialog creation predicates will return before the \texttt{e\_Create} clause or its default handler completes. The code which calls \texttt{win\_Create} can use this mechanism to indirectly pass data to the event-handler code by using the data directly as the \texttt{Creation Data}, if it is numeric and fits within a LONG. If not, more complex data can be passed by using \texttt{cast} to store a pointer to the data, and passing the pointer as the LONG value.

If there is no \texttt{e\_Create} clause, or if it fails, the default \texttt{e\_Create} handler will be called and will succeed - the window will always be created and displayed. If the initialization fails, ordinarily your program should supply a second \texttt{e\_Create} clause, which catches this situation, and take appropriate measures. The Code Expert cannot generate duplicate event clauses for you - they must be cut and pasted manually. Remember to remove the code-expert-generated early cut from all \texttt{e\_Create} clauses except the last.

\textbf{Note}: If a window is being destroyed during the \texttt{e\_Create} event by calling the predicate \texttt{win\_Destroy}, the predicate \texttt{win\_Create} will generate the error 6306.

Creating a Regular Window

\begin{verbatim}
WINDOW = win_Create(WINTYPE, 
                    Rct, Title, Menu, ParentWin, WSFlags, EventHandler, CreationData)
\end{verbatim}

\texttt{win\_Create} is used to create either Top-Level or Child windows. The parameter WINTYPE specifies which kind of window it should be. The \texttt{Rct} parameter specifies the size of the new client window, and \texttt{Title} will be displayed in the title bar of the window (if the window is specified to have a title bar). For Top-
Level windows the menu parameter can be used to specify which menu a window should have. *ParentWin* specifies the window handle of the parent window, which for Top-Level windows must be either the Screen window or the Task window, and for Child windows, it must be either a Top-Level window or another Child window. *WSFlags* is a list of flag values that specifies how the window should look and behave. *EventHandler* specifies the predicate, which should be called when an event arrives for the window. *CreationData* is a LONG value that will be passed to the event-handler predicate as part of the *e_Create* event.

**Creating a Dialog**

```
WINDOW = win_CreateResDialog(
    ParentWin, WINTYPE, ResId, Eventhandler, CreationData)
```

There are two kinds of dialogs: modeless and modal dialogs. A modal dialog must be satisfied before the user can activate any other window, menu or control in the application. On the other hand, any number of modeless dialogs can concurrently be on the screen of which any one can be made active by the user clicking the mouse in it so that it will accept input.

This predicate will create either a modal or a modeless dialog as described in the WINTYPE parameter.

**Creating a Control**

```
WINDOW = win_CreateControl(
    WINTYPE, Rct, Title, ParentWin, WinStyleFlags, CtrlId)
```

This predicate is used to add a new control to either a window or a dialog. (On some platforms it is not possible to add controls to a dialog after creation). WINTYPE is used to specify which kind of control to create. *CtlFlags* is a list of parameters to specify the behavior of the control. *CtrlId* is an integer constant used to identify the control within the window or dialog.

**Creating Windows Dynamically**

It is possible to specify either a window or dialog with included controls in a single Prolog structure from the WINDEF_LIST domain. This may indicate that all the information is to be obtained from the resource segment, or may specifically include all the controls and font information. Additionally, you can
specify all the coordinates in dialog base units, to make the dialogs independent of the font.

**GLOBAL DOMAINS**

```
WINDEF_LIST = WINDEF*
WINDEF =
  resdlg(RESID, WINTYPE);
dlg(WDEF, WSFLAGS);
dlg_font(WDEF, STRING FontName, Integer FontSize, WSFLAGS);
topwin(WDEF, MENU, WSFLAGS);
childwin(WDEF, CTLID, WSFLAGS);
icon(WDEF, CTLID, RESID, WSFLAGS);
ctl(WDEF, CTLID, WSFLAGS);
customctl(WDEF, CLASSNAME, CTLID, WSFLAGS)
```

```
WDEF = wdef(WINTYPE, RCT, STRING, UNIT_TYPE)
```

**UNIT_TYPE CONSTANTS**

```
u_Pixels = 0 % scaling in pixels
u_DlgPlatform = 1 % scaling in Platform Native Dialog Base Units
u_DlgBase = 2 % scaling in Windows Based Dialog Base Units
```

Windows and dialogs may be built dynamically during the application from a list of component controls, rather than using pre-built resources from the application's resource segment. The first element in the list must specify the window, and the rest of the elements in the list must specify either Child windows or controls. In every case the handle to the created window or dialog is returned.

To build and create a window:

```
WINDOW = win_CreateDyn(WINDEF_LIST, WINDOW Parent, EHANDLER, LONG CreationData)
```

To build and create a dialog:

```
WINDOW = win_CreateDynDialog(WINDOW Parent, WINDEF_LIST, EHANDLER, LONG CreationData)
```

To create a number of controls in a created window or dialog which may or may not yet be displayed:

```
WINDOW = win_CreateDynControl(WINDEF_LIST, WINDOW Parent)
```

This predicate returns the handle to the first of the created controls.
Destroying Windows

A window, a dialog or a control can be destroyed by a call to `win_Destroy`. When this call has been made for a particular window, the window's event-handler predicate will receive an `e_Destroy` event. This should be used to retract any facts, close files etc. associated with the window.

When a window is destroyed, all of its Child windows are automatically destroyed. All Child windows will receive an `e_Destroy` event before the parent window receives this event.

Example code for window destruction:

```prolog
% The OK button will close the window
mywin_event_handler(Win, e_Control(idc_ok, _, CtlWin, CtlInfo), 0):- !,
   win_Destroy(Win).

% Do all clean-up in the e_Destroy event
mywin_event_handler(Win, e_Destroy, 0):- !,
   retractall(mywin_data(Win, _)).
```

Retrieving the Client Rectangle

The `win_GetClientRect` function retrieves the coordinates of the window's Client Area. `win_GetClientRect` fills a RCT structure with the coordinates of the upper-left and lower-right corners of the client area, but the coordinates are relative to the Client Area itself (Client Coordinates). This means that the coordinates of a client area's upper-left corner are always (0, 0), and the coordinates of the lower-right corner are the width and height of the client area.

\[ \text{ClientRCT} = \text{win_GetClientRect}(\text{Window}) \]

The Clipping Rectangle

By default, window output is clipped to the client area, but it is possible to set the clipping region explicitly to any rectangle with a call to `win_SetClip`:

```prolog
my_event_handler(Win, e_Update(_Rct), 0):- !,
   win_SetClip(Win, rct(60, 60, 100, 100)),
   draw_Text(Win, 70, 70, "Hello, World").
```

All future output to the given window will then be clipped to the specified rectangle.
To stop clipping, just set clipping to a rectangle at least as large as the client rectangle.

The current clipping rectangle for any window may be obtained:

\[
\text{ClipRct} = \text{win\_GetClip(WINDOW)}
\]

In both cases, the rectangle coordinates are relative to the client rectangle of the window.

**Retrieving the Outer Rectangle**

An application can retrieve the coordinates of a window's outer rectangle by:

\[
\text{OuterRect} = \text{win\_GetOuterRect(Window)}
\]

The RCT structure is filled with the coordinates of the window's upper-left and lower-right corners. For dialogs and for the Task window the coordinates returned are relative to the screen, and for all other windows relative to the parent window.

**Calculating Client Rectangle from Outer Rectangle**

If a window must be placed on the basis of its outer rectangle (for example while moving or re-sizing operations), the predicate `rect\_GetClient` can be used to convert the desired outer rectangle to the resulting client rectangle. The styles for the window must be given as input:

\[
\text{ClientRCT} = \text{rect\_GetClient(WSFlags, BOOLEAN HasMenu, OuterRct)}
\]

In this case the `ClientRCT` RCT structure returns the window's client area coordinates relative to the parent window coordinates. (For dialogs and Task windows the returned RCT structure is relative to the screen coordinates.)

**Moving and Re-sizing Windows**

After a window is created, an application can change the window's size and position by calling `win\_Move`. The following example shows how to move a window 10 pixels to the right and down on a mouse click:
win_event_handler(Win, e_MouseUp(PNT, ShiftCtlAlt, Button), 0) :- !,
    OuterRCT = win_GetOuterRect(Win),
    WsFlags = win_GetState(Win),
    ClientRCT = rect_GetClient(WsFlags, const_HasMenu, OuterRCT),
    NewClient = rect_Offset(ClientRCT, 10, 10),
    win_Move(Win, NewClient).

Note that you cannot simply use rect_Offset on the rectangle returned by win_GetClientRect, because that rectangle is relative to itself, i.e., always has an origin of (0,0). See also Coordinate System section (page 90).

Changing the State for a Window

When a window is created, it is given a number of style flags. These flags indicate the initial state settings for a window. They determine whether the window is visible, disabled or minimized. A table of possible window styles can be found on page 9.

It is possible to change the state of a window by a call to win_SetState, and to receive the actual state for a window with a call to win_GetState.

    win_SetState(WINDOW, WSFLAGS WinStateList)
    WSFLAGS = win_GetState(WINDOW)

Disabling and Enabling Windows

A disabled window receives no keyboard or mouse input, but it can receive messages from other windows, from other applications, and from Windows. An application typically disables a window to prevent the user from using the window. For example, an application may disable the OK push button in a dialog box to prevent the user from choosing it until certain required data in other controls is present. An application can enable a disabled window at any time. A window is usually enabled when created, but the wsf_Disabled style can be specified in order to create a disabled window.

When a Child window is disabled, Windows passes the child's mouse input messages to the parent window as if there was no Child window covering the given rectangle.

To enable and disable windows use the calls:

    win_SetState(WINDOW, [wsf_Disabled])
    win_SetState(WINDOW, [wsf_Enabled])

Chapter 2, Windows
Hiding and Showing Windows

A window or control can be made totally invisible. In this case the user will see no effect of the window. However the program can still use the window as if it where a regular window. It is very common to let dialogs contain invisible controls, which become visible when some button or action is performed.

To hide and make visible windows use the calls:

\[
\text{win\_setState} (\text{WINDOW}, \text{[wsf\_Invisible]}) \\
\text{win\_setState} (\text{WINDOW}, \text{[wsf\_Visible]})
\]

Minimizing, Maximizing and Restoring Windows

A window can be minimized, maximized or restored to its previous size from one of these states. This can either be done by the user if the window is created with the proper style flags, or it can be done by calling the \text{win\_setState} predicate.

To minimize, maximize or restore windows use the calls:

\[
\text{win\_setState} (\text{WINDOW}, \text{[wsf\_Minimized]}) \\
\text{win\_setState} (\text{WINDOW}, \text{[wsf\_Maximized]}) \\
\text{win\_setState} (\text{WINDOW}, \text{[wsf\_Restored]})
\]

Changing the Icon Associated with a Window

To associate with a window the icon that is drawn in the window's title bar and when the window is minimized, use the predicate:

\[
\text{Win\_SetIcon} (\text{Window}, \text{IconResourceIdentifier})
\]

Changing the Window Text or Window Title

This predicate is used to change the text shown by a static text control and the text contained in an edit control:

\[
\text{win\_SetText} (\text{Window}, \text{Title})
\]

For windows that have a title bar, it is also the predicate used to change the text of the window's title (such windows do not have content text in the above sense).
Changing the Event-handler

The VPI allows you to dynamically change the event-handler predicate for a window. This is done using the predicate:

\[
\text{win\_SetHandler}(\text{Win}, \text{other\_event\_handler})
\]

All events for the specified window will subsequently be directed to the \text{other\_event\_handler} predicate. This feature is used as an example in the Tree tool when a tree should be put into an already created window.

Connecting Data to a Window

Data can be directly connected to a window. This feature is not used frequently, but it allows you to retrieve and change a LONG value relating to an internal window control structure. This LONG value can at any time be retrieved and changed with a call to the two predicates:

\[
\text{Data} = \text{win\_GetData}(\text{WINDOW}),
\]
\[
\text{win\_SetData}(\text{WINDOW}, \text{NewData})
\]

Finding the Task Window

Every running program has the task window. The handle of the task window can be obtained by a call to \text{vpi\_GetTaskWin}:

\[
\text{TaskWinHandle} = \text{vpi\_GetTaskWin}()
\]

Finding the Parent Window

Every window except the Screen window has a parent. The handle of the parent window of any given window can be obtained by a call to \text{win\_GetParent}:

\[
\text{ParentWin} = \text{win\_GetParent}(\text{ChildWin})
\]
Finding the Active Window

The active (foreground) window is either a Top-Level window or a task window that either has the focus itself or contains a control or Child window that has the focus. The handle to the active window can be returned with a call to:

```c
Win = win_GetActiveWindow()
```

Setting the Focus

Only one control or window at a time can have the focus. Having the focus means that it will receive input from the keyboard. Changing the focus can either be done by the user by clicking on another control, or by the application with a call to:

```c
win_SetFocus(NewWindow)
```

Finding the Window which has the Focus

Similarly, the reverse is a call to:

```c
Window = win_GetFocus()
```

This is used to find out which control or window currently has the focus.

Window Ordering

Sometimes it is necessary to bring a window to the front of the screen without changing the focus. This can be done with a call:

```c
win_BringToTop(Window)
```

Window Updating

The application must always be ready to re-draw the given rectangle of the window in response to an `e_Update(Rct)` event. If the window has been re-sized, restored, partly or fully invalidated or an overlapping window has been fully or partly removed, an `e_Update(Rct)` event will occur. For simplicity, the application can just re-draw the entire window and the windowing system will automatically clip the drawing to just the area where the updating is needed.
Since the application must be able to restore the window contents on an e_Update event, it is very important to arrange the application so that all window drawing is done here. This centralizes all window output in one place in the code, instead of having different portions produced at different events.

The following predicates are used to control when a window needs to be updated. The predicates can be used to optimize the drawing, so as little as possible will be redrawn.

**Invalidate Window**

\[
\text{win}_{-}\text{Invalidate} (\text{Window})
\]

\[\text{win}_{-}\text{Invalidate} \]

is the standard way for you to initiate total re-drawing of one of your windows because it immediately causes an e_Update event to be sent to that window's event-handler.

\[
\text{win}_{-}\text{Invalidate}(\text{Window}, \text{Rct})
\]

This predicate can be used to specify that only a part of the window needs to be updated.

**Validate Part of the Window**

\[
\text{win}_{-}\text{ValidateRect} (\text{Window}, \text{Rct})
\]

This is used to tell the windowing system that no updating is to be done in the specified part of the window.

**Check whether Part of the Window Needs Updating**

\[
\text{win}_{-}\text{NeedsUpdate} (\text{Window}, \text{Rct})
\]

The \[\text{win}_{-}\text{NeedsUpdate}\] can be used in certain situations to speed up window updating or printing. Before drawing an area that requires time-consuming calculations, database look up or complex drawing, this predicate can be used to test whether the indicated region really needs to be re-drawn. While the e_Update term will indicate one rectangle (corresponding, say, to the area just uncovered by another window), the updating code may need to segment its drawing by regions defined by its underlying data structures. By using \[\text{win}_{-}\text{NeedsUpdate}\] it can ask about specific sub-rectangles more convenient to its purposes.
Force Updating of the Window

\texttt{win\_Update} (Window)

The \texttt{win\_Update} predicate causes any update events that might be pending in the event queue to occur and ensures that the window is completely drawn. For example, this might be needed before scrolling a part of the window.
CHAPTER 3

Events

Events occurring in the VPI system result in the construction of a data structure from the EVENT domain according to the type of event. These event structures are sent to the event-handler for the window owning or originating the event and will appear as the second argument to the event-handler. Descriptions of the event domain structures and the recommended action by the event-handler for each of them follow. Where no clause exists for a particular event domain alternative, default handling is performed - the default handling is also shown.

To study the events sent in various situations, you should run the VPI example in the Events subdirectory.

Window Creation

e_Create(LONG CreationData)

This event is the first one that any window or dialog receives. It occurs just after the window or dialog has been internally created, but before it has been displayed. Generally, this is the right place to do initialization (such as database assertions, enabling, disabling, or graying controls, and initializing control values). You will note that the Code Experts in the VPI place their code for creating controls, Child windows, and tool and help bars here so that everything is created before the window becomes visible. The predicate that creates the window will not return until the e_Create event clause has completed.

It is frequently necessary to pass instance-specific information from the creation predicate through to the created instance of the window. A good example would be a new or modified title if the same dialog is used from more than one place in an application. The only place this can be done is in the event-handler's e_Create event, since the window and its window handle do not exist until that time and, since the dialog is not yet visible, changes can be made without being visible on the display.

This can be done by giving a value in the LONG CreationData parameter of the creation predicate; this value is then passed into the e_Create event. The code would look something like:
Note that the \texttt{e\_Create} event for the Task window is used as the place to do initialization code for the application as a whole.

**Window Being Destroyed**

\texttt{e\_Destroy()}

This event is the place to do the final retraction of all instance data that is specific to the window. It is the last event your window will process. Your code has access to all controls and Child windows (and their contents) even though they are not visible on the screen.

A window will always receive an \texttt{e\_Destroy} event, independently of whether the window was destroyed by a call to \texttt{win\_Destroy} or whether its parent window was destroyed.

**User Requests to Close Window**

\texttt{e\_CloseRequest()}

This event tells you that the user has double-clicked the system menu button for the window or selected \textit{Close} from its system menu. If you do not catch this event, the default handling is to destroy the window, without performing any application wind-up operations. If the event-handler predicate succeeds for this event, no automatic closing of the window is done. You might as an example catch this event to perform application wind up and/or to query the user and ask for confirmation before calling \texttt{win\_Destroy ()}.

**GUI is Closing Down**

\texttt{e\_EndSession(AbortPossible)}:
When this event is received, the GUI environment is about to shut down. It can be invoked either by the user or in some automatic way as in a power failure where the operating system has some time to terminate because of an UPS system. This message originates with the underlying operating system and is sent to all running applications.

The event is sent to the Task window event handler. In response to it the clause catching \texttt{e\_EndSession} event in the event handler predicate in your application should fail, if it approves the close request, or succeed if it disapproves.

The usual action taken for an \texttt{e\_EndSession} event is to save any modified data.

The return parameter, in the below arbitrarily set to \texttt{b\_True}, is meaningless.

\begin{verbatim}
my_window_event_handler(Win, e_EndSession(b_True), b_True):- !,
% If this code is executed and succeeds the system does not close.
% This predicate must fail if it is OK to close the system
\end{verbatim}

\section*{Menu Becomes Active}

\texttt{e\_InitMenu()}

The \texttt{e\_InitMenu} event is sent when a menu is about to become active. It occurs when the user clicks an item on the menu bar or presses a menu key. This allows the application to modify the drop-down menu before it is displayed.

\section*{A Menu Selection has been Made}

\texttt{e\_Menu(MENU\_TAG, INTEGER ShiftCtrlAlt)}

When the user makes a menu selection, an \texttt{e\_Menu} event is sent to the application. The parameter \texttt{MENU\_TAG} is the resource identifier of the constant that identifies the selected menu entry. The \texttt{ShiftControlAlt} parameter contains the state of the \texttt{Shift} and \texttt{Controls} keys that may have been pressed during activation of the menu entry.

\begin{verbatim}
task_win_event_handler(Win, e_Menu(id_file_exit, ShiftCtrlAlt), 0):- !,
  win_Destroy(Win).
\end{verbatim}

\texttt{e\_Menu} events are only sent to windows with enabled menus. If the window is created without a menu, it will not receive \texttt{e\_Menu} events. On some platforms, there are various restrictions concerning menus for Dialogs or Child windows.
The tool bar push buttons for a window will also send $e_{\text{Menu}}$ events to the window's event-handler.

Often windows need to pass unrecognized events to the Task window for default processing of things like File Handling, Help etc. The code for this is automatically inserted by the Code Expert for default handling of $e_{\text{Menu}}$ events:

```prolog
my_event_handler(W, e_Menu(ID, ShiftCtrlAlt), 0):- !,
    PW = win_GetParent(W),
    win_SendEvent(PW, e_Menu(ID, ShiftCtrlAlt)).
```

### Mouse Click

$e_{\text{MouseDown}}$(PNT, ShiftCtrlAlt, Button)

This event is generated when a mouse button has been clicked in the client area of the window.

The **Button** parameter indicates which button has been pressed. The following constants are defined in VPI.CON: $\text{mouse\_button\_left}$, $\text{mouse\_button\_right}$, $\text{mouse\_button\_middle}$.

**Note:** VPI refers to ‘left button’ and ‘right button’ for simplicity. For users operating the mouse with their left hand, the buttons are reversed. This is handled by the operating system, so developers can safely assume that the ‘left button’ is the primary mouse button, while the ‘right button’ is the secondary button.

The **ShiftControlAlt** parameter contains the status of the Shift, Control and Alt keys. The following constants are defined in VPI.CON: $c_{\text{Nothing}}$, $c_{\text{Shift}}$, $c_{\text{Control}}$, $c_{\text{Alt}}$, $c_{\text{ShiftCtl}}$, $c_{\text{ShiftAlt}}$, $c_{\text{CtlAlt}}$, $c_{\text{ShiftCtlAlt}}$.

The PNT structure contains the position in the client window where the mouse click occurred.

Normally you do not invoke actions on an $e_{\text{MouseDown}}$ event, instead this event is used to prepare an operation that should react to or track $e_{\text{MouseMove}}$ events until finally the $e_{\text{MouseUp}}$ event occurs. To make sure there will be an $e_{\text{MouseUp}}$ message, you need to Capture the mouse within the window; otherwise, the mouse could be moved outside of the window before it is released. In that case no $e_{\text{MouseUp}}$ events would be sent to the original window's event-handler.
When the mouse is clicked on buttons, menus, borders etc. other control events are generated, which are not *Mouse* events.

**Mouse Double Click**

\[ e_{\text{MouseDbl}}(\text{PNT, ShiftControlAlt, Button}) \]

This event is generated when a mouse button is double clicked in the client area of the window. The *MouseDbl* event does not preclude the constituent *e_MouseDown* and the *e_MouseUp* messages also being sent.

**Mouse Moved in Window**

\[ e_{\text{MouseMove}}(\text{PNT, ShiftCtrlAlt, ButtonsList}) \]

This event is generated whenever the mouse is moved inside a window, and PNT represents the new location. A stream of these events will be generated while the mouse moves.

*MouseMove* events can be used for various purposes. For example, it can be used to handle dragging such as when the VP text editor drags a block of text. Yet another frequent usage is to make a rubber band-like line for marking an area - this is done in the VP dialog editor.

**Mouse Button Released**

\[ e_{\text{MouseUp}}(\text{PNT, ShiftCtrlAlt, Button}) \]

This event is generated anytime a mouse button is released in the client area of the window.

**Note:** An *e_MouseUp* event is not always preceded by an *e_MouseDown* event. Your application must be able to ignore such situations.

**Keyboard Input**

\[ e_{\text{Char}}(\text{INTEGER Char, INTEGER ShiftCtlAlt}) \]

When a key is pressed on the keyboard, an *e_Char* event will be sent to the window, which currently has the focus.
There are basically two different key types: the data characters that occur in text, and the control keys like Enter, Del, the function and the cursor keys. The data keys are represented by their value, while the control keys are indicated by virtual key codes defined in VPI.CON.

**A Key has been Pressed Down**

\[ e\_KeyDown(\text{INTEGER Char}, \text{INTEGER ShiftCtrlAlt}) \]

This event is sent as soon as a key is pressed.

**A Key has been Released**

\[ e\_KeyUp(\text{INTEGER Char}, \text{INTEGER ShiftCtrlAlt}) \]

This event is sent when a key is released.

\texttt{e\_KeyDown} and \texttt{e\_KeyUp} events usually occur in pairs, but if the user holds down a key long enough to start the keyboard’s automatic repeat feature, the system generates a number of \texttt{e\_Char} events in a row. It then generates a single \texttt{e\_KeyUp} event when the user releases the key. So the following sequence of events is generated:

\[ e\_KeyDown, \texttt{e\_Char, e\_Char, ...}, \texttt{e\_Char, e\_KeyUp} \]

**Horizontal Scroll Event**

\[ e\_HScroll(\text{SCROLLCODE, INTEGER Pos}) \]

This event is sent when the user activates the horizontal scroll bar of a window. The \texttt{Pos} parameter indicates the setting for the scroll bar.

**Vertical Scroll Event**

\[ e\_VScroll(\text{SCROLLCODE, INTEGER Pos}) \]

This event is sent when the user activates the vertical scroll bar of a window. The \texttt{Pos} parameter indicates the setting for the scroll bar.
Focus is being Gained

e_GetFocus

This event occurs before focus is given to the window so the user has not been able to press any buttons or to use any other controls that your window can have.

The e_Control(_,_,_,getfocus) event comes to a dialog or a window if its edit or list edit controls gain focus.

Focus is being Lost

e_LoseFocus

This event is generated when the user selects another window or application that will cause the current window to lose focus. When the event arrives the window still has the focus, and keeps it until your code has been executed.

The e_Control(_,_,_,losefocus) event comes to a dialog or a window if its edit or list edit control loses focus.

Background Erasing

e_EraseBackground()

The e_EraseBackground is sent when a window needs to be updated. Normally the windowing system fills a window prior to updating with the default background color. If the event-handler predicate succeeds for this event, no background painting is done.

If you want a transparent window, or you are sure that you draw the background anyway as part of processing the e_Update event, you can catch this event (make it succeed) to make updating faster and free of flicker.

% Do no automatic background filling
my_window_event_handler(Win, e_EraseBackground(), 0):- !.

Window Location has Changed

e_Move(TopCornerX, TopCornerY)
This event tells you that the position of your window has changed. The top left hand corner of your window's client rectangle has been moved to position \textit{TopCornerX, TopCornerY} (in screen coordinates).

An \textit{e\_Move} event can be forced by using the \textit{win\_Move} predicate.

\textbf{Window has Changed Size}

\texttt{e\_Size(\text{NewWidth, NewHeight})}

This event tells you that the window has changed size. If you have saved the old size in a database (maybe you did this in the \textit{e\_Create} event) you can determine whether the window has been made larger or smaller than before. In some cases, the window will not need to be updated if it has been made smaller.

The usual action taken in response to an \textit{e\_Size} event is to recalculate the size of drawings, Child windows or controls and adjust or move their size and positions. As an example, if some text is to remain always centered in a window, the entire window needs to be updated, the technique for this is to invalidate the window by a call to \textit{win\_Invalidate}, where the code for the update event would always calculate a new center and redraw the window.

Note that these events can either be triggered by the user re-sizing or moving the window or by a call to \textit{win\_Move}. They are also sent when a window is minimized or maximized.

\textbf{Window has Changed State}

\texttt{e\_State(StateInfo)}

The \textit{e\_State} event is sent immediately after the state of a task window, top-level window, child window or custom control has changed. The \textit{StateInfo} notifies about the following circumstances:

\begin{tabular}{|l|l|}
\hline
\textit{StateInfo} & Description \\
\hline
\texttt{font(FONT)} & \texttt{win\_SetFont} is called \\
\hline
\texttt{text(STRING)} & \texttt{win\_SetText} is called \\
\hline
\texttt{disabled()} & window is disabled by \texttt{win\_setState} \\
\hline
\texttt{enabled()} & window is enabled by \texttt{win\_setState} \\
\hline
\texttt{invisible()} & window is set invisible by \texttt{win\_setState} \\
\hline
\end{tabular}
visible() window is set visible by \textit{win\_SetState}

\section*{Native GUI Events}

\texttt{e\_Native(Message, wParam, lParam)}

The \texttt{e\_Native} event provides the possibility of receiving all native GUI events that are sent to a window. To enable such \texttt{e\_Native} messages, you must call:

\begin{verbatim}
win_EnableHook(WINDOW, BOOLEAN OnOff)
\end{verbatim}

with a \texttt{b\_True} parameter.

If the \texttt{e\_Native} event-handler succeeds, no default VPI processing is done for the event. If it fails, the event will if needed be converted to the usual VPI event.

A LONG value can be returned to the windowing system.

\section*{Owner Drawing}

\texttt{e\_OwnerDraw(CtlType, CtlId, ItemID, Action, State, Window, RCT RectItem, LONG ItemData)}

This event is sent to a window, which has controls with the \texttt{wsf\_OwnerDraw} or \texttt{wsf\_OwnerDrawVariable} style flag. When this event is received, it is up to the application to draw the shape of a control.

For further information see the section about Owner Drawing (Page 62), and the OWN\_DRAW example in the VPI\TOOLEXAM directory.

\section*{Return Owner Draw Size}

\texttt{e\_OwnerMeasureItem(CtlType, CtlId, ItemId, Data)}

This event is sent to a window, which has created some controls with the \textit{OwnerDraw} style flag. The event is used to let the application specify the size of a Menu Item, a List Item in a list box etc. The size must be returned as the return value of the event-handler predicate.

\section*{Updating the Window}

\texttt{e\_Update(RCT)}
When your window receives this event you know that the system wants you to redraw the window. The rectangle RCT specifies the coordinates for the part of your window that needs to be updated. Often it is easier to redraw the whole window than to work out what to draw in the part that has been invalidated. Either way, the windowing system sets up a clipping region so that whatever you draw, only the invalid area is actually updated.

```
my_window_event_handler(Win, e_Update(ClipRct), 0):- !,
    % Get Client Rectangle
    ClientRCT = win_GetClientRect(Win),
    ClientRCT = rct(Wl, Wt, Wr, Wb),
    : % Your Update code goes here
    :;
```

The e_Update event can be triggered by using the win_Invalidate predicate.

**Timer Interval has Expired**

```
e_Timer(LONG TimerId)
```

Following the issue of:

```
TimerId = timer_Set(Window, Interval)
```

**e_Timer** events will be sent to the event-handler for Window every Interval milliseconds until timer_Kill(TimerId) is issued. The **TimerId** can be used to distinguish between multiple timers.

For further information about timers see the Timer section (page 118) and the RUNTIMER example in the VPI\EXAMPLES directory.

**A Programmed Event has Occurred**

```
e_User(LONG Id, LONG Ptr)
```

The **e_User** events are a way to make application-specific extensions to the event domain.

**Dynamic Data Exchange**

```
e_DDE(DDE_CONV Conv, DDE_EVENT Event)
```
When doing Dynamic Data Exchange (DDE) under MS Windows, the Task window event-handler automatically receives all the e_DDE events. The first parameter is a handle to the particular conversation going on, the second is a substructure, which further specifies the event.

For further information about the DDE handling see the section about DDE (page 136) and the DDE example in the VPI\EXAMPLES directory.

Control Notification Events

$e_{\text{Control}}(\text{CtlId, CtlType, CtlWin, CONTROL_INFO})$

Controls send events to their parent window using $e_{\text{Control}}$ events. These events are normally notification events from the control. Notification events include those for getting and losing focus (e.g. using the Tab key to move out of an edit control), clicking on a button, changing values or a scroll bar etc.

$\text{CtlId}$ is the integer constant assigned during resource editing that uniquely identifies the control in the window or dialog.

$\text{CtlType}$ is the WINTYPE of the control and will be a 'wc_' prefixed constant.

$\text{CtlWin}$ is the window handle of the control.

CONTROL_INFO is a sub-message that explains the activation cause in detail. The domain alternatives are described below.

The CONTROL_INFO Domain

Events for Scroll bars

$\text{scroll(ScrlEvent, Pos)}$

Where $\text{Pos}$ is the new position of the scroll bar thumb tracker box on the scroll bar (after the user has moved it).

$\text{ScrlEvent}$ identifies the type of scroll bar event:

- $\text{Sc\_None}$: nowhere (event should be ignored)
- $\text{Sc\_LineUp}$: one line up
- $\text{Sc\_LineDown}$: one line down
- $\text{Sc\_PageUp}$: previous page
Sc_PageDown  next page
sc_Top          to top
sc_Bottom       to bottom
sc_Thumb        thumb re-positioning
sc_ThumbTrack   dynamic thumb tracking

The Control is Gaining focus
getfocus()

The Control is Losing focus
losefocus()

Control Content has been Modified.
modified()

This occurs once for every keystroke or mouse operation, which results in a change to the value displayed in the control.

Events for List boxes
selchanged()

dropdown()  
dropdown will be sent for list edits and list buttons when the user drops down the list. The event can be used to temporarily disable heavy operations on selchanged events.

closeup()  
dropdown will be sent for list edits and list buttons when the user closes the dropdown list.

Control is activated by mouse or keyboard
activated()
Application Termination Notification Events

\[ e_{\text{EndApplication}}(\text{ApplicationId}) \]

VPI sends this event to the event-handler of the \textit{ReceiveMsgWin} window immediately after termination of an application (with the application identifier \textit{ApplicationId}) that was run by a call:

\[ \text{ApplicationId} = \text{vpi}_{\text{CreateProcess}}(\text{ReceiveMsgWin}, \text{Command}, \ldots, \ldots) \]

Here the \textit{Command} specifies the program to be loaded.

Application has Activated

\[ e_{\text{Activate}}() \]

VPI sends \textit{e_Activate} event to the task window event-handler immediately after the application has activated (has become foreground).

Application has been Deactivated

\[ e_{\text{Deactivate}}() \]

VPI sends \textit{e_Deactivate} event to the task window event-handler immediately after the application has been deactivated.
CHAPTER 4

Controls

Controls are specialized Child windows. They are mainly used inside dialog boxes to perform input/output, but they can also be created inside a regular window. It is not possible to draw in controls (unless ownerdrawing is specified), and it is not possible to supply an event-handler for a control. Their events cause notification messages to be sent to their parent window event-handler. The predefined control classes are listed below:

Table Error! Bookmark not defined..3: Predefined Control Classes

<table>
<thead>
<tr>
<th>Control type:</th>
<th>Window class:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push button</td>
<td>wc_PushButton</td>
</tr>
<tr>
<td>Radio Button</td>
<td>wc_RadioButton</td>
</tr>
<tr>
<td>Check box</td>
<td>wc_CheckBox</td>
</tr>
<tr>
<td>Hscroll</td>
<td>wc_Hscroll</td>
</tr>
<tr>
<td>Vscroll</td>
<td>wc_Vscroll</td>
</tr>
<tr>
<td>Edit</td>
<td>wc_Edit</td>
</tr>
<tr>
<td>Text</td>
<td>wc_Text</td>
</tr>
<tr>
<td>Lbox</td>
<td>wc_Lbox</td>
</tr>
<tr>
<td>LBoxButton</td>
<td>wc_LboxButton</td>
</tr>
<tr>
<td>LBoxEdit</td>
<td>wc_LboxEdit</td>
</tr>
<tr>
<td>Group Box</td>
<td>wc_GroupBox</td>
</tr>
<tr>
<td>Icon</td>
<td>wc_Icon</td>
</tr>
</tbody>
</table>
Manipulating Controls

Because controls are windows, an application can manipulate them by using the window-management functions. The window handle for a control can always be retrieved from the parent window and the resource identifier for the control:

\[
\text{CtrlHndl} = \text{win\_GetCtlHandle(ParentWin, CtrlId)}
\]

The following table shows a number of predicates that are often used for the controls.

\[\text{Table} \text{ Error! Bookmark not defined.}.4: \text{Manipulating Controls}\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Predicate</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation</td>
<td>\textit{win_CreateControl}</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>\textit{win_CreateDynControl}</td>
<td>49</td>
</tr>
<tr>
<td>Destruction</td>
<td>\textit{win_Destroy}</td>
<td>17</td>
</tr>
<tr>
<td>Size/Location</td>
<td>\textit{win_GetOuterRect}</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>\textit{win_Move}</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>\textit{win_GetClientRect}</td>
<td>17</td>
</tr>
<tr>
<td>Title/Label/Text</td>
<td>\textit{win_GetText}</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>\textit{win_SetText}</td>
<td>50</td>
</tr>
<tr>
<td>Show, Hide, Enable, Disable</td>
<td>\textit{win_GetState}</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>\textit{win_setState}</td>
<td>19</td>
</tr>
<tr>
<td>Focus</td>
<td>\textit{win_GetFocus}</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>\textit{win_SetFocus}</td>
<td>22</td>
</tr>
<tr>
<td>Window hierarchy</td>
<td>\textit{win_GetParent}</td>
<td>21</td>
</tr>
<tr>
<td>Window type</td>
<td>\textit{win_GetType}</td>
<td>8</td>
</tr>
</tbody>
</table>

Style Flags

When creating a control, you can specify various style flags for the control. Some style flags are portable, while others exist only on specific platforms. However, it is possible on all platforms to specify any flag - inappropriate flags will simply be ignored. In this way it is possible to create both portable applications, which at the same time take advantage of all possibilities on each
platform. The meaning of each flag when present in the list of flags is covered in the table below.

### Table Error! Bookmark not defined.5: General Control Style Flags

<table>
<thead>
<tr>
<th>Flag Identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsf_Disabled</td>
<td>The control will be initially disabled</td>
</tr>
<tr>
<td>wsf_Checked</td>
<td>Start the check box in the checked state</td>
</tr>
<tr>
<td>wsf_Default</td>
<td>Flag for default push button in a dialog</td>
</tr>
<tr>
<td>wsf_Invisible</td>
<td>Create an Invisible control</td>
</tr>
<tr>
<td>wsf_Group</td>
<td>Flag to start a new group of controls</td>
</tr>
<tr>
<td>wsf_TabStop</td>
<td>Allow tabbing to the control</td>
</tr>
<tr>
<td>wsf_ReadOnly</td>
<td>A read-only edit control</td>
</tr>
<tr>
<td>wsf_MultiLine</td>
<td>Allow multiple lines in an edit control</td>
</tr>
<tr>
<td>wsf_MultiSelect</td>
<td>Allow multiple selections in a list box</td>
</tr>
<tr>
<td>wsf_MultiColumn</td>
<td>Specify multiple columns in a list box</td>
</tr>
<tr>
<td>wsf_AlignLeft</td>
<td>Left alignment of text</td>
</tr>
<tr>
<td>wsf_AlignCenter</td>
<td>Centered alignment of text</td>
</tr>
<tr>
<td>wsf_AlignRight</td>
<td>Right alignment of text</td>
</tr>
<tr>
<td>wsf_LeftText</td>
<td>Specifies that the check box or radio button should have the text on the left side.</td>
</tr>
<tr>
<td>wsf_OwnerDraw</td>
<td>Specifies ownerdrawing for a control, in this case the user must do all the drawing of the control</td>
</tr>
<tr>
<td>wsf_VScroll</td>
<td>Specifies that a multi-line edit control or a list box will get a vertical scroll bar when it needs one</td>
</tr>
<tr>
<td>wsf_HScroll</td>
<td>Specifies that a multi-line edit control or a list box will have a horizontal scroll bar when it needs one</td>
</tr>
<tr>
<td>wsf_NoBorder</td>
<td>Specifies that there should not be a border around the control</td>
</tr>
<tr>
<td>Control</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>wsf_Simple</td>
<td>Designates a simple rectangle and displays a single line of text left-aligned in the rectangle. The line of text cannot be shortened or altered in any way</td>
</tr>
<tr>
<td>wsf_NoWrap</td>
<td>Designates a simple rectangle and displays the given text left-aligned in the rectangle. Tabs are expanded but words are not wrapped. Text that extends past the end of a line is clipped</td>
</tr>
<tr>
<td>wsf_NoPrefix</td>
<td>Prevents interpretation of any 'e' characters in the control's text as accelerator prefix characters (which are displayed with the 'e' removed and the next character in the string underlined). This static control style may be included with any of the defined static controls. This is most often used when filenames or other strings that may contain an e need to be displayed in a static control in a dialog box</td>
</tr>
<tr>
<td>wsf_UpperCase</td>
<td>Converts all characters to upper case as they are typed into the edit control</td>
</tr>
<tr>
<td>wsf_LowerCase</td>
<td>Converts all characters to lower case as they are typed into the edit control</td>
</tr>
<tr>
<td>wsf_Password</td>
<td>Displays all characters as an asterisk (*) as they are typed into the edit control</td>
</tr>
<tr>
<td>wsf_AutoVScroll</td>
<td>Automatically scrolls text up when the user presses Enter on the last line</td>
</tr>
<tr>
<td>wsf_AutoHScroll</td>
<td>Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the Enter key, the control scrolls all text back to position zero</td>
</tr>
<tr>
<td>wsf_NoHideSel</td>
<td>Negates the default behavior for an edit control. The default behavior is to hide the selection when the control loses the input focus and invert the selection when the control receives the input focus</td>
</tr>
</tbody>
</table>

*Chapter 4, Controls*
<p>| wsf_OEMConvert | Converts text entered in the edit control from the Windows character set to the OEM character set and then back to the Windows set. This ensures proper character conversion when the application calls the AnsiToOem function to convert a Windows string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames |
| wsf_WantReturn | Specifies that a carriage return will be inserted as data when the user presses the <strong>Enter</strong> key while entering text into a multi-line edit control in a dialog box. If this style is not specified, pressing the <strong>Enter</strong> key has the same effect as pressing the dialog box's <strong>Default</strong> push button. This style has no effect on a single-line edit control |
| wsf_3State | Creates a button that is the same as a check box, except that the box can be grayed (dimmed) as well as checked. The grayed state is used to show that the state of the check box is not determined |
| wsf_Auto | Automatic radio button groups or automatic check box |
| wsf_Auto3State | Automatic 3-State check box |
| wsf_NoNotify | Do not notify the parent window with an input message whenever the user clicks or double-clicks in a list box |
| wsf_Sort | Sorts strings in the list box alphabetically |
| wsf_NoReDraw | Specifies that the list box’s appearance is not updated when changes are made. This style can be changed at any time by using <strong>lbox_Suspend</strong> and <strong>lbox_ReSume</strong> |</p>
<table>
<thead>
<tr>
<th>wsf_OwnerDrawVariable</th>
<th>Specifies that the owner of the list box is responsible for drawing its contents and that the items in the list box are variable in height. The owner window receives a <code>e_OwnerMeasureItem</code> message for each item in the list box when the list box is created and a <code>e_OwnerDrawItem</code> message whenever the visual aspect of the list box changes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsf_HasStrings</td>
<td>Specifies that a list box contains items consisting of strings. By default, all list boxes except ownerdrawn list boxes have this style. An application can create an ownerdrawn list box either with or without this style.</td>
</tr>
<tr>
<td>wsf_UseTabStops</td>
<td>Allows a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog box units.</td>
</tr>
<tr>
<td>wsf_NoIntegralHeight</td>
<td>Specifies that the size of the list box is exactly the size specified by the application when it created the list box. Normally, Windows re-sizes a list box so that the list box does not display partial items.</td>
</tr>
<tr>
<td>wsf_WantKeyboardInput</td>
<td>Specifies that the parent window of the list box receives WM_VKEYTOITEM or WM_CHARTOITEM messages whenever the user presses a key and the list box has the input focus. This allows an application to perform special processing on the keyboard input. If a list box has the LBS_HASSTRINGS style, the list box can receive WM_VKEYTOITEM messages but not WM_CHARTOITEM messages. If a list box does not have the LBS_HASSTRINGS style, the list box can receive WM_CHARTOITEM messages but not WM_VKEYTOITEM messages.</td>
</tr>
<tr>
<td>wsf_ExtendedSel</td>
<td>Allows multiple items to be selected by using the Shift key and the mouse or special key combinations.</td>
</tr>
</tbody>
</table>
A table showing which 'wsf_' flags are meaningful for each class of control under Windows and which are meaningful under OS/2 Presentation Manager follows:

<table>
<thead>
<tr>
<th>Control</th>
<th>Possible Flags Under Windows</th>
<th>Possible Flags under PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push button</td>
<td><em>wsf_Disabled</em>, <em>wsf_Default</em>, <em>wsf_Invisible</em>, <em>wsf_Group</em>, <em>wsf_TabStop</em>, <em>wsf_OwnedDraw</em></td>
<td><em>wsf_Disabled</em>, <em>wsf_Default</em>, <em>wsf_Invisible</em>, <em>wsf_Group</em>, <em>wsf_TabStop</em>, <em>wsf_NoBorder</em></td>
</tr>
<tr>
<td>Control Type</td>
<td>Possible Attributes</td>
<td>Example Attributes</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Check box</td>
<td><code>wsf_Disabled</code>, <code>wsf_Invisible</code>, <code>wsf_Group</code>, <code>wsf_TabStop</code>, <code>wsf_OwnerDraw</code>, <code>wsf_LeftText</code>, <code>wsf_Auto</code>, <code>wsf_3State</code>, <code>wsf_Auto3State</code></td>
<td><code>wsf_Disabled</code>, <code>wsf_Invisible</code>, <code>wsf_Group</code>, <code>wsf_TabStop</code>, <code>wsf_Auto</code>, <code>wsf_3State</code></td>
</tr>
<tr>
<td>Radio button</td>
<td><code>wsf_Disabled</code>, <code>wsf_Invisible</code>, <code>wsf_Group</code>, <code>wsf_TabStop</code>, <code>wsf_LeftText</code>, <code>wsf_OwnerDraw</code>, <code>wsf_Auto</code></td>
<td><code>wsf_Disabled</code>, <code>wsf_Invisible</code>, <code>wsf_Group</code>, <code>wsf_TabStop</code>, <code>wsf_Auto</code></td>
</tr>
<tr>
<td>Static text</td>
<td><code>wsf_Disabled</code>, <code>wsf_Invisible</code>, <code>wsf_AlignLeft</code>, <code>wsf_AlignCenter</code>, <code>wsf_AlignRight</code>, <code>wsf_BlackRect</code>, <code>wsf_GrayRect</code>, <code>wsf_WhiteRect</code>, <code>wsf_BlackFrame</code>, <code>wsf_GrayFrame</code>, <code>wsf_WhiteFrame</code>, <code>wsf_Simple</code>, <code>wsf_LeftNoWordWrap</code>, <code>wsf_NoPrefix</code></td>
<td><code>wsf_Disabled</code>, <code>wsf_Invisible</code>, <code>wsf_AlignLeft</code>, <code>wsf_AlignCenter</code>, <code>wsf_AlignRight</code>, <code>wsf_BlackRect</code>, <code>wsf_GrayRect</code>, <code>wsf_WhiteRect</code>, <code>wsf_BlackFrame</code>, <code>wsf_GrayFrame</code>, <code>wsf_WhiteFrame</code>, <code>wsf_LeftNoWordWrap</code>, <code>wsf_NoPrefix</code></td>
</tr>
<tr>
<td>Group box</td>
<td><code>wsf_Disabled</code>, <code>wsf_Invisible</code></td>
<td><code>wsf_Disabled</code>, <code>wsf_Invisible</code></td>
</tr>
<tr>
<td>Control</td>
<td>Additional Properties</td>
<td>Additional Properties</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Edit control</td>
<td>wsf_Disabled, wsf_Invisible, wsf_Group, wsf_TabStop,</td>
<td>wsf_Disabled,</td>
</tr>
<tr>
<td></td>
<td>wsf_AlignLeft,</td>
<td>wsf_Invisible,</td>
</tr>
<tr>
<td></td>
<td>wsf_AlignCenter,</td>
<td>wsf_Group,</td>
</tr>
<tr>
<td></td>
<td>wsf_AlignRight,</td>
<td>wsf_TabStop,</td>
</tr>
<tr>
<td></td>
<td>wsf_MultiLine,</td>
<td>wsf_AlignLeft,</td>
</tr>
<tr>
<td></td>
<td>wsf_AutoVScroll,</td>
<td>wsf_AlignCenter,</td>
</tr>
<tr>
<td></td>
<td>wsf_AutoHScroll,</td>
<td>wsf_AlignRight,</td>
</tr>
<tr>
<td></td>
<td>wsf_HScroll,</td>
<td>wsf_MultiLine,</td>
</tr>
<tr>
<td></td>
<td>wsf_Password,</td>
<td>wsf_AutoVScroll,</td>
</tr>
<tr>
<td></td>
<td>wsf_NoHideSel,</td>
<td>wsf_AutoHScroll,</td>
</tr>
<tr>
<td></td>
<td>wsf_OEMConvert,</td>
<td>wsf_NoBorder(for multiline),</td>
</tr>
<tr>
<td></td>
<td>wsf_WantReturn,</td>
<td>wsf_ReadOnly,</td>
</tr>
<tr>
<td></td>
<td>wsf_NoBorder,</td>
<td>wsf_Password</td>
</tr>
<tr>
<td></td>
<td>wsf_UpperCase,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wsf_LowerCase,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wsf_ReadOnly</td>
<td></td>
</tr>
<tr>
<td>Scroll bar</td>
<td>wsf_Disabled,</td>
<td>wsf_Disabled,</td>
</tr>
<tr>
<td></td>
<td>wsf_Invisible,</td>
<td>wsf_Invisible,</td>
</tr>
<tr>
<td></td>
<td>wsf_Group,</td>
<td>wsf_Group,</td>
</tr>
<tr>
<td></td>
<td>wsf_TabStop</td>
<td>wsf_TabStop</td>
</tr>
<tr>
<td>List box</td>
<td>wsf_Disabled,</td>
<td>wsf_Disabled,</td>
</tr>
<tr>
<td></td>
<td>wsf_Invisible,</td>
<td>wsf_Invisible,</td>
</tr>
<tr>
<td></td>
<td>wsf_Group,</td>
<td>wsf_Group,</td>
</tr>
<tr>
<td></td>
<td>wsf_TabStop</td>
<td>wsf_TabStop</td>
</tr>
<tr>
<td></td>
<td>wsf_MultiSelect,</td>
<td>wsf_MultiSelect,</td>
</tr>
<tr>
<td></td>
<td>wsf_ExtendedSel,</td>
<td>wsf_ExtendedSelect,</td>
</tr>
<tr>
<td></td>
<td>wsf_OwnerDraw,</td>
<td>wsf_ExtendedSel,</td>
</tr>
<tr>
<td></td>
<td>wsf_OwnerDrawVariable,</td>
<td>wsf_ExtendedSel,</td>
</tr>
<tr>
<td></td>
<td>wsf_HasStrings,</td>
<td>wsf_OwnerDraw,</td>
</tr>
<tr>
<td></td>
<td>wsf_NoBorder,</td>
<td>wsf_OwnerDraw,</td>
</tr>
<tr>
<td></td>
<td>wsf_Sort,</td>
<td>wsf_NoBorder,</td>
</tr>
<tr>
<td></td>
<td>wsf_NoNotify</td>
<td>wsf_Sort,</td>
</tr>
<tr>
<td></td>
<td>wsf_MultiColumn,</td>
<td>wsf_NoNotify,</td>
</tr>
<tr>
<td></td>
<td>wsf_HScroll,</td>
<td>wsf_MultiColumn,</td>
</tr>
<tr>
<td></td>
<td>wsf_VScroll,</td>
<td>wsf_HScroll,</td>
</tr>
<tr>
<td></td>
<td>wsf_NoRedraw</td>
<td>wsf_VScroll,</td>
</tr>
<tr>
<td></td>
<td>wsf_UseTabStops,</td>
<td>wsf_NoRedraw,</td>
</tr>
<tr>
<td></td>
<td>wsf_WantKeyboardInput,</td>
<td>wsf_UseTabStops,</td>
</tr>
<tr>
<td></td>
<td>wsf_DisableNoScroll</td>
<td>wsf_WantKeyboardInput,</td>
</tr>
<tr>
<td></td>
<td>wsf_NoIntegralHeight</td>
<td>wsf_DisableNoScroll,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wsf_NoIntegralHeight</td>
</tr>
<tr>
<td>Control Type</td>
<td>Win32 Flags</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>List edit</td>
<td>wsf_Disabled, wsf_Invisible, wsf_Group, wsf_TabStop, wsf_OwnerDraw, wsf_OwnerDrawVariable, wsf_HasStrings, wsf_Sort, wsf_VScroll, wsf_NoIntegralHeight, wsf_AutoHScroll, wsf_DisableNoScroll, wsf_OEMConvert</td>
<td>wsf_Disabled, wsf_Invisible, wsf_Group, wsf_TabStop, wsf_HScroll</td>
</tr>
<tr>
<td>List button</td>
<td>wsf_Disabled, wsf_Invisible, wsf_Group, wsf_TabStop, wsf_OwnerDraw, wsf_OwnerDrawVariable, wsf_HasStrings, wsf_Sort, wsf_VScroll, wsf_NoIntegralHeight, wsf_AutoHScroll, wsf_DisableNoScroll</td>
<td>wsf_Disabled, wsf_Invisible, wsf_Group, wsf_TabStop, wsf_HScroll</td>
</tr>
<tr>
<td>Icon</td>
<td>wsf_Disabled, wsf_Invisible, wsf_Group, wsf_TabStop, wsf_Topmost, wsf_Transparent, wsf_HScroll, wsf_Vscroll</td>
<td></td>
</tr>
<tr>
<td>Custom control</td>
<td>wsf_Disabled, wsf_Invisible, wsf_Group, wsf_TabStop, wsf_Topmost, wsf_Transparent, wsf_HScroll, wsf_Vscroll</td>
<td>wsf_Disabled, wsf_Invisible, wsf_Group, wsf_TabStop</td>
</tr>
</tbody>
</table>

**Events from Controls**

Even though controls are windows like any others, control event-handlers are not ordinarily part of an application, since standard controls are usually implemented by the host windowing system itself, or in extensions to it in the case of custom controls. When a control is created, the control must be the Child of another window. This relationship is important because a control sends messages, called notification messages, to its parent window when events such as input from the user occur in the control. The application uses these notification messages to invoke the appropriate processing. The notification messages appear as parent window **e_Control** events, which include the Control's resource identifier, the type of the control, the Window handle of the control and provision for extra information describing the event:
e_Control(INTEGER Id, WINTYPE, Window Ctrl, CONTROL_INFO);

The extra information for the control event belongs to the following domain:

CONTROL_INFO =
    activated();                    % Click on a button etc.
    scroll(SCROLLCODE, INTEGER);   % Kind of scroll + new position
    getfocus();                    % Control will get focus
    losefocus();                   % Control will lose focus
    modified();                    % Content has been modified
    dropdown();                    % Listbutton drop down of list
    closeup();                     % Listbutton closes up of list
    selchanged;                    % Selection changed in listbox

Note: e_Control events are not sent for scroll bars which are created according to the window style flags along the edges of a window. For these, special e_VScroll and e_HScroll messages are sent.

Creating Controls

Creation of controls can either be automatic as part of creating a dialog from a description defined in the resource segment of an application or they may be created dynamically at run time. Dialogs in the resource files are laid out by the dialog editor in the VDE, and they are then created at run time by a call to win_CreateResDialog. Dialogs not laid out in resource files can be created dynamically at run time by a call to win_CreateDynDialog, but all parameters must then be specified at creation time.

To create a new control in a window, the predicate win_CreateControl can be used. To create a number of controls in a window from a structure definition (a WINDEF_LIST) win_CreateDynControl can be used. It is possible to add extra controls to a regular window or a dialog. One way to work around this is to create an invisible control, which can be made visible later.

The possibilities for creating a control can be seen by studying the parameters to the win_CreateControl predicates:
The Various Controls

Group Box Controls

A group box is a rectangle in a dialog - it can appear as depressions on the dialog surface. The left-aligned title replaces the top line from the left for the length necessary. A group box is used to indicate visually that the set of controls within it has some common meaning as a group. A group box has no functional effect - it does not actually invoke group behavior in the controls it surrounds (this is done using the \texttt{wsf\_Group} style for the controls in question), rather it is just like a static text control, it cannot be selected and no events are sent. There is no parent-child relationship to the controls inside a group box.

Icon Controls

An icon control allows you to display in a window or dialog an icon defined in the resources. They are always small bitmaps that are mapped to the pixels of the display. The apparent size relative to their parent window may therefore vary with screen resolution. They behave like a static text control in that no events are sent.

When an icon from the resource segment must be created dynamically, call \texttt{win\_CreateDynControl} with the ICON domain alternative:

\begin{verbatim}
  icon(WDEF, CTLID, RESID, WSFLAGS);
\end{verbatim}
**Static Text Controls**

Text controls contain small pieces of text, generally used as labels for other controls, which can be placed at specific locations in the window or dialog. A static text control does not generate any events. It is possible to change the text while the application is running by a call to `win_SetText` and to hide and show the control by `win_SetState`.

**Edit Controls**

Edit controls are used to allow the user to input text. These are often just a single line, which will contain a name or a number etc., but can also be much larger and support multiple lines of data.

To set and get the content of an edit control the following predicates must be used:

```
STRING = win_GetText(WINDOW)
win_SetText(WINDOW, STRING Text)
```

Where WINDOW refers to the window handle for the control (which can be obtained by a call to:

```
CtrlHndl = win_GetCtlHandle(ParentWin, CtrlId)
```

**Edit controls send three notification events:**

```
ehandler(Win, e_Control(CtlID, CtrlType, CtrlWin, getfocus), 0):-
ehandler(Win, e_Control(CtlID, CtrlType, CtrlWin, losefocus), 0):-
ehandler(Win, e_Control(CtlID, CtrlType, CtrlWin, modified), 0):-
```

The modified notification is issued synchronously with every change made by the user, and will therefore be issued many times if the text in the edit control is being edited or if new text is being entered.

**Push Buttons Controls**

Push buttons are used to let the user invoke actions and are very simple to use. They can be either the usual **OK**, **Close**, or **Cancel** buttons to close the window or dialog, or other buttons that spawn a new window or dialog, start printing etc.

Push buttons are controls that appear to recede on mouse down and return on mouse up. The mouse up generates an `e_Control` event with the resource identifier of the push button, which is sent to the event-handler of the parent
window. If the mouse is dragged outside the push button before being released, the push button will return but no event will be generated.

For keyboard operation of push buttons, use the **Tab** or **Shift+Tab** keys to navigate until the desired push button is highlighted, then press the space bar.

If the resource identifier of the control is `idc_hello`, the following clause could be used to create a little pop-up note:

```prolog
ehandler(Win, e_Control(idc_hello, _, _, _), 0):- !,
    dlg_Note("Hello There!").
```

**Default Push Button**

A push button can be assigned to be the default push button for a dialog. This means that if **Enter** is pressed anywhere in the dialog (except within an edit control with `wsf_WantReturn` set), the dialog will behave as if the Default push button had been pressed.

A Default push button is initially drawn with a heavier surrounding rectangle, indicating that it is the active control. A default push button is created by using the style flag `wsf_Default`, this is normally done by checking the corresponding option for a Default push button in the dialog editor in the VDE.

**Notification Events for Push Buttons**

Push buttons generate only one type of event:

```prolog
ehandler(Win, e_Control(CtlId, CtlType, CtlWin, activated), 0):-
```

Tabbing to or away from a push button will not generate `getfocus` or `losefocus` `e_Control` events.

**Check Box Controls**

Check boxes are used to indicate a Boolean state. As an example, the check boxes in the VP compiler options dialog provide various settings indicating whether stack checking needs to be done, line number information should be generated etc. Unlike radio buttons, each check box is normally independent of other check boxes.

When the user clicks in a check box, an `e_Control` event is sent to the event-handler predicate for the window. If you are not using the Dialog Package, then you need to write code for your application to change the state of the check box. This can be accomplished with a call to the predicate:
A clause to handle the reversal of the check mark could look like the following:

```
mywin_event_handler(Win, e_Control(idc_check, CtrlType, CtrlWin, _), 0) :- !, !,
% Check that it is valid to change the state
State = win_IsChecked(CtrlWin),
toggle(State, NewState),
win_Check(CtrlWin, NewState).
```

When using the Dialog Package, you do not need to worry about changing the state of check boxes. In this case the initial setting must be supplied when the dialog is created and the final resulting value is returned when the dialog is terminated.

**Note:** Under MS Windows it is also possible to have a check box that automatically reverses its state by setting the flag `wsf_Auto`.

**Check Provide Notification of Only One Type of Event:**

```
ehandler(Win, e_Control(CtlId, CtlType, CtlWin, activated), 0):-
```

There are no focus notifications sent when tabbing to or away from a check box.

**Radio Button Controls**

Radio buttons are similar to check boxes with the difference that only one of the buttons in a group of radio buttons can be checked.

On the MS Windows platform, it is possible to give the `wsf_Auto` flag for radio buttons, which means that when the user clicks on a new radio button in a group, Windows itself will turn off the previously checked button in the group.

However for portability reasons, there is a predicate:

```
win_CheckRadioButton(WINDOW, WINLIST RadioWinList)
```

which can be used to check a new button, and will automatically turn off the previous selection in the group specified by the list of radio buttons. When using the Dialog Package, this predicate is used to manage the radio button checking automatically.
Notification Events from Radio Buttons
Radio buttons generate only one event:

\[ e\text{handler} (\text{Win}, \text{e\_Control}(\text{CtlId, CtlType, CtlWin, activated}), 0):= \]

There are no focus notifications sent when the user uses the TAB key to move the focus to or away from a check box.

List Box Controls
A list box is a control that contains a list of items from which the user can choose. If the list box contains more elements than can be displayed simultaneously in the window, and the wsf\_VScroll flag is not turned off, the window will automatically provide a vertical scroll bar. The user can scroll up and down in the list to change a current selection or double-click on a given list box element.

There are various specialized list boxes like list buttons and list edits, and by setting appropriate flags you can allow multiple selections in a list box or multiple columns (multiple columns are not supported on all platforms).

List boxes will normally contain strings, but by using the ownerdraw style, you can create list boxes that can display any kind of data.

List Box Predicates
The VPI has a number of predicates, which operate on list boxes. These all have the prefix "lbox\_". Each of these predicates requires the handle to the list box, which can be obtained with a call to \text{win\_GetCtlHandle} using the handle to the parent window and the list box resource identifier. When specifying an index in a list box, 0 indicates the first, 1 the second etc. The index value -1 has a special meaning - it is used to add an element at the end.

The VPI\EXAMPLES\LISTBOX example shows how to use these list box predicates:

\text{lbox\_Add(WINDOW, INTEGER Index, STRING Str)}

This predicate will add a new string to the list box and display it if it lies within the list box window. It will be inserted so the new string will be positioned at Index after it is inserted. If \text{Index} is -1 the string will be added to the end of the list and displayed if still in the list box window. No scrolling will occur and the currently selected string remains selected.
**lbox\_Add(WINDOW, INTEGER Index, SLIST StringList)**

This version of *lbox\_Add* adds all the strings in the list *StringList* list at the given position in the existing list. Using the *Index* -1 will add the list of strings to the end of the list box.

**lbox\_Add(WINDOW, STRING Str)**

This version must be used when the sort attribute is set for a list box. For a list box with no sorting the items will be added to the end of the list box.

**lbox\_Add(WINDOW, SLIST StringList)**

This version must be used to add a list of strings to a list box with the sort flag set. If sort is not set, the strings will be added to the end of the list box.

**lbox\_Clear(WINDOW)**

Calling this predicate will clear the list box and remove all entries.

**lbox\_Delete(WINDOW, INTEGER Index)**

This predicate can be used to delete a given element in the list. 0 is the first, 1 is the second etc. A -1 value will be ignored.

**SLIST = lbox\_GetAll(WINDOW)**

Returns all the strings from a list box as a list of strings.

**INTEGER = lbox\_CountAll(WINDOW)**

Returns the number of strings in the list box.

**lbox\_GetItem(WINDOW, INTEGER Index)**

This predicate can be used to retrieve a given element in the list. 0 is the first, 1 is the second etc. A -1 value will be ignored.

**lbox\_GetSel(WINDOW, SLIST, ILIST)**

Returns a list of the selected strings and a list of their indices, both lists in the same order. This predicate is intended for multiple selection list boxes, although
it will return single item lists for regular list boxes with something selected. If there are no selections, the predicate will return two empty lists.

\[ \text{INTEGER} = \text{lbox\_GetSelIndex}(\text{WINDOW}) \]

Return the index of the current selection. The predicate will fail if there is no current selection.

\[ \text{lbox\_IsSel}(\text{WINDOW, INTEGER Index}) \]

This predicate will succeed or fail depending on whether or not the given element in a list box is currently selected.

\[ \text{lbox\_SetSel}(\text{WINDOW, INTEGER Index, BOOLEAN Select}) \]

This predicate is used to select or de-select an element in a list box. For a multiple selection list box, this predicate will turn on or off the selection of this one element without affecting other selections. For single selection list boxes, setting the selection to a given element will automatically turn off any previous selection.

\[ \text{lbox\_Suspend}(\text{WINDOW}) \]
\[ \text{lbox\_Resume}(\text{WINDOW}) \]

These two predicates can be used to inhibit screen updating temporarily and then restore it, as might be required to keep a list box from 'flashing' during updating activity. As an example, if filling a list box from an external data-source, involving a series of fetch-data - \text{lbox\_Add} operations, it will be more visually pleasing to enclose the operation with \text{lbox\_Suspend, lbox\_Resume}.

\[ \text{lbox\_SetColumnWidth}(\text{WINDOW, INTEGER width}) \]

This predicate can be used to set the width of the columns in multi-column list boxes.
The preceding dialog was created by selecting the multi-column style for the list box, and adding the following code for the initialization:

```prolog
dlg_choose_your_favorite_language_eh(_Win, e_Create(_),0):-!,
   LB = win_getCtlHandle(_Win, idc_choose_your_favorite_language),
   lbox_setColumnWidth(LB, 56),
   !.
```

**lbox_SetTabStops**(WINDOW, ILIST TabList)

For a list box defined with the flag *wsf_UseTabStops*, you can use this predicate to set tab stops to define "visual columns". The textual information presented on each line of the list box here is considered one element. Note that this is not to be confused with the multi-column list boxes. If the list box strings have embedded tab characters, the characters following each tab will be aligned with the corresponding position specified in the tab position list. The tab positions are given in Dialog Base Units.

This technique is required if the dialog font is proportional and you need to align columns of information as in the following display:
Note that each line remains a single selected entity, as the tab stop affects only the line's display characteristics.

As an example of calling the predicate, the above tab settings were initialized as:

```prolog
dlg_select_employee_eh(_Win, e_Create(_),0):-!,
    List = win_getCtlHandle(_Win, idc_select_employee_1),
    lbox_setTabstops(List, [36,128,238]),
    !.
```

The actual tab stop values were derived by subtracting the dialog units value of the list control's left-most coordinate from the leftmost coordinate of each of the text controls forming the headers.

**Notification Events from List Boxes**

% when the user makes a selection
```
ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, selchanged), 0):-
```

% when the user double clicks in a list box
```
ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, activated), 0):-
```

% when a list box is about to gain focus
```
ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, getfocus), 0):-
```

% when a list box is about to lose focus
```
ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, losefocus), 0):-
```
**lbox_SetTopIndex(WINDOW Win, INTEGER Index)**

Scrolls the list box so that either the *Index* item appears at the top of the list box or the maximum scroll range has been reached. The *Index* is the zero-based index of the item in the list box.

**List Buttons**

List buttons are a combination of a push button and a list box. In the initial state the list button occupies a single line, but there is a little push button on the right which when pressed shows the contents of the list in a pop-up window below or, failing that, above the control, with a vertical scroll bar if necessary. A selection can be done from the list of alternatives. Pressing the button again will retract the list from the display, leaving the current selection in the remaining single line. Both the single line and the list are read-only and cannot be edited by the user.

The styles, events and operations are the same as for list boxes, with the following exceptions:

When creating a list button, the rectangle given will be the size of the control including the drop down list.

A list button cannot have the multi-selection style flag. List buttons also do not support tab stops or multiple columns, so the tab stop and multi-column flags, as well as the *lbox_SetTabStops* and *lbox_setColumnWidth* predicates are not applicable.

**Notification Events from List Buttons**

% when the user changes a selection
  
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, selchanged, 0):-

% when the List-button drops down
  
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, dropdown), 0):-

% when the List-button closes up
  
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, closeup), 0):-

% when a List-button is about to gain focus (only under Windows)
  
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, getfocus), 0):-

% when a List-button is about to lose focus (only under Windows)
  
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, losefocus), 0):-
List Edit Boxes

A List Edit is a combination of an edit control and a list box. They behave primarily like edit controls, but the user can pop up a list of predefined strings and if one is selected, it is placed in the edit control for further editing. With the list showing, editing the single line can be used to edit that selection. With the list showing, clearing the single line and entering a new value can be used to place a new value in the list in addition to the old one. The drop down list cannot be edited by the user (although you could add this feature yourself).

Notification Events from List Edit Boxes

% when changing the selection:
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, selchanged, 0), 0):-

% when the List-button drops down
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, dropdown), 0):-

% when the List-button closes up
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, closeup), 0):-

% when a List-button is about to gain focus
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, getfocus), 0):-

% when a List-button is about to lose focus
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, losefocus), 0):-

% when a List-button is modified
  ehandler(Win, e_Control(CtrlID, CtrlType, CtrlWin, modified), 0):-

Scroll Bars

With the window types wc_HScroll and wc_VScroll, it is possible to create horizontal and vertical scroll bars at any location in a window. They behave like the scroll bars belonging to the window decorations, and can be used in a very similar way.

Scroll bars can also be specified as part of edit controls and of list box controls. In these cases the scroll processing is done by the control and no notification is provided to the parent dialog.

Notification Events from Scroll Bars

Scroll bar controls send e_Control events, while the scroll bars belonging to the window decorations send e_HScroll and e_VScroll events.
The event-handler clauses for these three types of scroll bars are:

\[
\text{ehandler}(\text{Win}, \text{e\_Control}(\text{ResId}, \text{CtrlType}, \text{CtrlWin}, \text{scroll}(\text{ScrCode}, \text{Pos})), 0) : - \\
\text{ehandler}(\text{Win}, \text{e\_VScroll}(\text{ScrCode}, \text{Pos}), 0) : - \\
\hspace{1cm} \% \text{ vertical scroll bar activated} \\
\text{ehandler}(\text{Win}, \text{e\_HScroll}(\text{ScrCode}, \text{Pos}), 0) : - \\
\hspace{1cm} \% \text{ horizontal scroll bar activated}
\]

\text{ScrCode} belongs to the domain \text{SCROLLCODE}, which tells what the user has done with the scroll bar (line up, page down, end etc.). Values are listed in the table below.

A single click on the scroll bar above or below the scroll thumb will generate a \text{sc\_PageUp} or \text{sc\_PageDown} event, respectively. It is entirely up to the programmer to interpret what these events should mean in the context of the application and make any resulting changes to the window.

The \text{Pos} field is important when the user drags the scroll bar thumb. This value indicates the thumb position relative to the range of the scroll bar. While the user is dragging the thumb, the application will receive \text{sc\_ThumbTrack} events, and when the user releases the thumb \text{sc\_Thumb} event will be sent.

\textit{Table Error! Bookmark not defined..7: Scroll Bar Codes}

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{sc_LineUp}</td>
<td>One line up</td>
</tr>
<tr>
<td>\text{sc_LineDown}</td>
<td>One line down</td>
</tr>
<tr>
<td>\text{sc_PageUp}</td>
<td>Previous page</td>
</tr>
<tr>
<td>\text{sc_PageDown}</td>
<td>Next page</td>
</tr>
<tr>
<td>\text{sc_Top}</td>
<td>To top</td>
</tr>
<tr>
<td>\text{sc_Bottom}</td>
<td>To bottom</td>
</tr>
<tr>
<td>\text{sc_Thumb}</td>
<td>Thumb re-positioning</td>
</tr>
<tr>
<td>\text{sc_ThumbTrack}</td>
<td>Dynamic thumb tracking</td>
</tr>
</tbody>
</table>
Scroll Bar Referencing

The same set of predicates is used to manipulate the scroll bars for the window decorations and the scroll bars, which are controls. Since there is no separate window handle for the decoration scroll bars, the handle to the window itself is used to reference both of these, and a special parameter SCROLL_TYPE is used to tell which scroll bar is referred to:

GLOBAL DOMAINS

    SCROLL_TYPE = INTEGER

CONSTANTS

    sb_Horz = 0 % Horizontal window-scroll bar
    sb_Vert = 1 % Vertical window-scroll bar
    sb_Ctl = 2 % Scroll bar controls in dialog or window

Scroll Bar Range

When working with the actual position of the scroll bar thumb, the range of the scroll bar must first be set. The scroll bar range can be set by a call to the predicate:

    win_SetScrollRange(WIN, SCROLL_TYPE, INTEGER Min, INTEGER Max)

and the range can be retrieved later by a call to:

    win_GetScrollRange(WIN, SCROLL_TYPE, INTEGER Min, INTEGER Max)

If you do not explicitly set a range, the default scroll range for the standard window decoration scroll bars is 0 to 100 (there is no default range for scroll bars used as controls). On the 16-bit Windows platform, the difference between Min and Max must be no larger than 32,767.

Scroll Bar Positioning

The actual position of the scroll bar thumb must be moderated by the application - it is not changed automatically by the user's mouse manipulations. The position must be inside the set range for the scroll bar. An application sets the position using:

    win_SetScrollPos(WINDOW, SCROLL_TYPE, INTEGER Pos)

The current position may be read using:

    INTEGER = win_GetScrollPos(WINDOW, SCROLL_TYPE)
Scroll Bar Proportioning

On some platforms it is possible to set the size of the thumb relative to the size of the scroll bar. This provides a visual representation of how much of the data can be seen in the window compared to the full data. The proportion is a value between zero and the length of the interval for the scroll bar (Max - Min)

\[
\text{win\_SetScrollProportion(WINDOW, SCROLL\_TYPE, UNSIGNED Proportion)}
\]

The current proportion may similarly be read using

\[
\text{UNSIGNED = win\_GetScrollProportion(WINDOW, SCROLL\_TYPE)}
\]

Owner Drawing for Controls

By specifying the attribute \texttt{wsf\_OwnerDraw} or \texttt{wsf\_OwnerDrawVariable}, it is possible to create a control which otherwise behaves as the selected control normally does, being responsible for the actual drawing of the control. Specifying this attribute means that the control's parent window event-handler (for dialog controls, the dialog event-handler) must be able to handle the following two events:

\[
\text{e\_OwnerMeasureItem(CtlType, CtlId, ItemId, LONG data);}
\]

\[
\text{e\_OwnerDraw(UNSIGNED CtlType, UNSIGNED CtlId, UNSIGNED ItemID, OD\_ITEMACTION ItemAction, OD\_ITEMSTATE ItemState, WINDOW Win, RCT RectItem, LONG ItemData);}\]

The first event is used for list boxes to measure the size of an item, the event-handler must return a value, which is the size of the control. The second is the signal for the application to draw the content of the control.

Please study the VPI\TOOLEXAM\OWN\_DRAW example for the details.

Custom Controls

Besides the portable built-in controls on most platforms, it is also possible to create and use non-portable custom controls.

Custom controls are normally distributed as Dynamic Link Libraries (.DLL), in this case you must have a .LIB file for the custom control, link it into the .EXE file, - and know the function to call to initialize the .DLL.
If you are creating a custom-control (Child) window programmatically, you would ordinarily use `win_CreateDynControl` in the following manner:

```c
CustomCtl = win_CreateDynControl(
    [customctl(wdef(wc_Custom, RCT, Title, u_Pixels),
             ControlClass, 101, FLAGS)], Win, 0),
```

The ControlClass is a string, which identifies the custom window class for the control.

When you want to put a custom control in a dialog from the VDE dialog editor, you need to select Custom from the Controls menu. In the properties of the control you need to type in the desired class name.

For custom controls, the Flags list must be formatted according to a special convention to allow simultaneous specification of native (custom-control specific) attribute values, which are the actual 32-bit values OR-ed together, and the VPI style values, which are multi-platform inclusive, and re-mapped by the VPI code to the underlying platform 32-bit values. By convention the Flags style list values are processed as VPI style constants until a 0 list element is encountered, following which all subsequent style values are directly OR-ed together. As an example, the previous FLAGS variable, when using one of the more popular grid controls in a situation where it is to start invisible, might be specified as:

```c
FLAGS=[ wsf_Tabstop, wsf_Invisible, % vpi flags
       0, % switch to non-mapped flags
       HGRID_ColumnSelect, % these are defined by header files
       HGRID_AllowInsert ], % supplied with controls
```

It is also possible to create dynamic dialogs using custom controls, as in the following call which uses some of Borland's popular control extensions:
Note that custom controls do not fully participate in the VPI framework. Thus, if they send non-standard messages to the dialog event-handler, these events will arrive as \texttt{eNative} events, which must be decoded according to control-specific constants or other information supplied by the vendor for use with these products.

In general, you must be sure that the custom control classes are properly initialized before attempting to use them in a dialog. The event-handlers specific to these controls are normally implemented in .DLL files. It is often necessary to call one or more initialization functions in the .DLL to initialize the control classes. Some DLLs accomplish this in a hidden way that depends on linking the DLL's import library in a particular order. If you are using a third party control library and you encounter an error when attempting to use your dialog, you should review the control's documentation to be sure that you are properly calling any documented initialization functions first, and as a last resort review your Build scripts to ensure that the DLL's import library (usually a file with the same name and a .LIB extension) is present, where it may need to precede the other libraries.

VBX controls are a more complicated extension of the same idea, where the control classes not only need to be initialized, but the host environment (in this case your Visual Prolog program) must pretend to be the Visual Basic Runtime environment, from the standpoint of properly following the message protocols expected by the VBX controls. Therefore the use of VBX controls requires that you link with an emulation library (such libraries ship with Visual C, Borland C, or can be purchased separately from third party vendors). Note that apart from
the special initialization requirements, the VPI programming is the same for these and for the simpler custom controls.

Note that custom controls can also be created in Visual Prolog by a call to `class_Create`. This is described in the section about handling window classes, see page 132.
Dialog boxes are a special kind of windows. Dialogs are normally filled with controls, which can interact with the user by displaying output or accepting input.

The VPI provides a number of "common dialogs", or convenient pre-defined dialog boxes and handlers with a simple function-call interface for commonly occurring situations. All VPI predicates in this group start with the prefix 'dlg_', and can be easily inserted into the source code using the right-button mouse menu.

Where no suitable common dialog exists, a custom dialog may be designed with the VDE Dialog editor. This dialog may be stored is the resource segment (.RES file) or be created dynamically by calling some predicates that builds up the dialog either in a single call from a WINDEF_LIST, or control by calling the predicate win_CreateControl. Whichever method is used, the VDE Dialog Expert can generate the code framework.

Setting, retrieving or changing the contents of the controls in a dialog can be done on a per-control basis by using VPI predicates, and can also be done on a per-dialog basis by using the VPI Dialog Package. The Dialog Package is described in chapter Error! Bookmark not defined. of this book.

One feature that is supported in dialogs but not in windows is the ability to navigate between controls by using the Tab and Shift+Tab keys.

There are two kinds of dialogs: modeless and modal dialogs. A modal dialog requires that certain actions be taken before the user can activate any other window, menu or control in the application. One way to remember this is to think of a modal dialog as putting the whole application into a particular mode - one in which the dialog must finish before anything else in the application can receive input. In contrast, any number of modeless dialogs can concurrently be on the screen of which any one can be made active by the user clicking the mouse in it so that it will accept input.

On the MS Windows platform, dialogs will always appear and remain in front of their parent window and do not respect its boundary.
Common Dialogs

Common dialog boxes make it easier and faster to develop applications. A common dialog box is a dialog box an application can display by calling a single predicate rather than by supplying a dialog box event-handler and a resource description containing a dialog box layout. The calls to the common dialogs can be pasted into the application being edited by using the Edit | Insert | Predicate call | Window, Dialog or Toolbar facility.

Get User Response to Yes/No etc. Question

\[
\begin{align*}
\text{Integer} &= \text{dlg\_Ask(PromptString, ButtonTitleList)} \\
\text{Integer} &= \text{dlg\_Ask(TitleString, PromptString, ButtonTitleList)} \\
\text{Integer} &= \text{dlg\_Ask(ParentWin, TitleString, PromptString, ButtonTitleList)}
\end{align*}
\]

The predicate \texttt{dlg\_Ask} puts up a little dialog, which has up to three buttons. The titles for the buttons are given in a list. The buttons will normally be something like \texttt{Yes}, \texttt{No} and \texttt{Cancel}. The possible responses correspond to the order in the \texttt{ButtonTitleList} and indicate which button was pressed:

\[
\begin{align*}
\text{resp\_default} &= 0 \\
\text{resp\_2} &= 1 \\
\text{resp\_3} &= 2
\end{align*}
\]

As example the following call:

\[
\_\text{ANSWER} = \text{dlg\_Ask("MyTitle", "MyPrompt",} \\
\text{["MyButton1", "MyButton2", "MyButton3"]},
\]

Will result in the following dialog:

![Dialog Example](image-url)
User Selects a Filename(s) to Opening/Saving a File(s)

STRING /*FirstFileName=*/ dlg_GetFileName (  
    STRING InitMask, SLIST Types, Title, ILIST Flags, StartPath,  
    SLIST SelectedFileNames)

The `dlg_GetFileName` is used to set up a dialog where the user can select a filename(s) for the file to open/save. The `InitMask` is the default file mask. The `Types` is a list of alternative file masks that the user can choose to select instead. The `Title` specifies the dialog title. The `Flags` is a list of flags specifying the dialog properties. The `StartPath` specifies a start directory for searching files. If `StartPath = ""`, the current directory is used. The `FirstFileName` returns the fully qualified filename of the first selected file.

The `dlg_GetFileName` allows the user to select a list of filenames when the `dlgfn_MultiSel` flag is specified in the `Flags` list. The `SelectedFileNames` is the output parameter containing the list of all selected filenames. If the flag `dlgfn_Save` is specified, the "Save As" style dialog box is displayed. If the selected file already exists, then the "Save As" style dialog box generates a message box in that the user must confirm whether to overwrite the file. If the `dlgfn_Save` flag is not specified, the "Open File" style dialog box is displayed (default).

As an example, consider this call:

```
Flags = [dlgfn_MultiSel],
FirstFileName = dlg_GetFileName("*.pro",  
    ["Prolog Files (*.pro)", "*.pro",  
    "Prolog Headers (*.pre)", "*.pre",  
    "All Files (*.*)", "*.*", ], "MyTitle",  
    Flags, "",  
    OutFileNames),
```

It will result in the following dialog:
User Input of a String

String = dlg_GetStr(Title, MessageString, InitString)
String = dlg_GetStr(ParentWin, Title, MessageString, InitString)

The \texttt{dlg\_GetStr} is used to let the user input a string. The following call:

\_NewSTRING = dlg\_GetStr("MyTitle","MyMessage","InitStr"),

This will result in the following dialog:

User Selects from a List of Strings

BOOLEAN = dlg\_ListSelect(Title, StringList, PreSelectionIndexInteger, SelectedString, IndexInteger)

BOOLEAN = dlg\_ListSelect(ParentWin, Title, StringList, PreSelectionIndexInteger, SelectedString, IndexInteger)
BOOLEAN = dlg_ListSelect(ParentWin, Title, StringList, PreSelectionIndexInteger, SelectedString, IndexInteger, BOOLEAN Sort)

The `dlg_ListSelect` is used to let the user choose an element from a list of values. As an example, see the following call:

```plaintext
SLIST=["First string", "Second String", "Third string"], 
PreSel= 1,
dlg_ListSelect("MyTitle",SLIST, PreSel, _StrSel, _Index),
```

This call will result in the following dialog:

![Dialog Example](image)

**User Selection and Set up of a Printer**

`dlg_PrintSetup(NoOfCopiesInteger, FirstPageInteger, LastPageInteger)`

The `dlg_PrintSetup` is used to let the user choose which printer to use for the printing. The call will return how many copies the user wants, and the desired page range.

```plaintext
dlg_PrintSetup(_NoOfCopies, _FirstPage, _LastPage),
```
This call will result in the following dialog:

![Print dialog](image)

### User Selection of a Color

COLOR = dlg_ChooseColor(OldColor)
COLOR = dlg_ChooseColor(ParentWin, OldColor)

The `dlg_ChooseColor` are used to let the user select a color.

InitColor = color_Red,
_NewCOLOR = dlg_ChooseColor(InitCOLOR),

This will result in the following dialog:

![Color dialog](image)
The \textit{dlg\_ChooseColor/2} is used to specify the window \textit{ParentWin} that will receive the focus, after the dialog will be closed.

\section*{User Selection of a Font}

\begin{verbatim}
NewFontBinary = dlg\_ChooseFont(OldFontBinary)

NewFontBinary = dlg\_ChooseFont(
    FONT OldFont,
    STRING FontName,
    INTEGER FontSize)
\end{verbatim}

The \textit{dlg\_ChooseFont} is used to let the user select a font.

\begin{verbatim}
OldFONT = font\_Create(ff\_fixed,[],9),
_NewFONT = dlg\_ChooseFont(OldFONT),
\end{verbatim}

This will result in the following dialog:

\begin{center}
\includegraphics[width=0.5\textwidth]{font_dialog.png}
\end{center}

\section*{Show an Error}

\begin{verbatim}
dlg\_Error(MessageString)
\end{verbatim}

The \textit{dlg\_Error} should be called when the program needs to display an error message. As example the following call:

\begin{verbatim}
dlg\_Error("My Error Message"),
\end{verbatim}
Will result in the following dialog:

![Error dialog]

**Show a Note**

```plaintext
dlg_Note(MessageString)
dlg_Note(TitleString, MessageString)
```

`dlg_Note` is handy for popping up a small note box to display a short message. As example the following call:

```plaintext
dlg_Note("MyTitle", "Hello There"),
```

Will result in the following dialog:

![Note dialog]

**User Defined Dialogs**

**Control Identifiers**

Each control in a dialog must have an integer identifier value that is unique within the dialog. This identifier is called the control identifier (`CtrlId`). The integer corresponding to the control identifier is used in notification events back to the parent window. It control identifier indicates which control issued the event.

When a new control is placed in a dialog with the VDE Dialog and Window editor, it receives a named Prolog constant identifier. At compilation time, the VDE generates a `<project>.CON` file, which assigns integer values to all the
control resource identifiers according to certain rules. This file is included in all
modules that utilize the VPI. At project build time, the VDE also generates the
<project>.RES file in which all controls are identified only by these integers.

**Special Control Resource identifiers**
The following resource identifiers should be used for all **OK** and **Cancel**
buttons. The constants are predefined in the VPI.CON file.

```
idc_ok = 1
idc_cancel = 2
idc_help = 3
```

When using the Dialog Package it is important that a dialog have these three
push buttons with the proper identifiers, because the Dialog Package assumes
that these push buttons exist, and uses these to terminate the dialog.

**Creating a Dialog**
If a dialog has been created with Visual Prolog Dialog Editor and is placed in the
resources segment, it is possible to create a dialog with the call:

```
WINDOW = win_CreateResDialog(
    ParentWin, WINTYPE, ResId, EventHandler, CreationData)
```

*ParentWin* should be the window to gain focus after the dialog is terminated.
*WINTYPE* should be either *wd_Modal* or *wd_ModeLess* depending on whether
a modal or modeless dialog should be created. *ResId* is the integer that identifies
the dialog in the .RES file, *EventHandler* is the Prolog callback predicate, which
is used to handle the events from the dialog, *CreationData* is a long value which
is passed to the *e_Create* event.

Alternatively, it is possible to create a dialog from a Prolog list structure by a
call to:

```
WINDOW = win_CreateDynDialog(ParentWin, WINDEF_LIST, EventHandler,
CreationData)
```

Another way to create a dialog is to first make a call to *win_Create* to create the
dialog window itself, and then create the controls one by one with calls to
*win_CreateControl*. 

74  Visual Programming Interface
Initializing a Dialog

As soon as the dialog is created, there will be sent an `e_Create` event to the event-handler for the dialog. In the clause catching the `e_Create` event, all controls in the dialog should be initialized. For each control type there are different predicates that are used to set the values of the control.

*Table Error! Bookmark not defined..8: Manipulation Predicates for Controls*

<table>
<thead>
<tr>
<th>Control type:</th>
<th>Manipulation predicates:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check boxes</td>
<td><code>win_Check</code></td>
</tr>
<tr>
<td>Scroll bars</td>
<td><code>win_SetScrollPos</code></td>
</tr>
<tr>
<td>Edit controls</td>
<td><code>win_SetText</code></td>
</tr>
<tr>
<td>Static text</td>
<td><code>win_SetText</code></td>
</tr>
<tr>
<td>List boxes</td>
<td>All the <code>lbox_</code> predicates</td>
</tr>
</tbody>
</table>

Note, that all the manipulation predicates work on the window handles for the controls, not on their control identifiers.
CHAPTER 6

Drawing Operations

There are a number of VPI predicates that draw text, pixels and various geometric shapes. These predicates are all prefixed by 'draw_'. Drawing operations in a window utilize the current drawing attributes for that window, which are referred to as drawing tools. The drawing tools are: Pen, Brush, Drawmode, Font, Foreground Color, Background Color and Background Filling Mode.

The example in VPI\EXAMPLES\DRAWING shows various combinations of drawing primitives and drawing tool options, and illustrates the corresponding results.

Drawing Tools

It is possible to retrieve the settings of all the current drawing tools for a window into a single variable and then use this variable to restore them at a later point, by using the two predicates \texttt{win\_GetDrawTools/1} and \texttt{win\_SetDrawTools/2}. Here is the code sequence:

\begin{verbatim}
SaveTools = win_GetDrawTools( Win ),
% Change the drawing settings and draw with different settings
win_SetDrawTools(Win, SaveTools ), % restore the pre-existing ones
\end{verbatim}

The sub-components of the drawing tools object are defined in the DRAWTOOLS domain:

\begin{verbatim}
GLOBAL DOMAINS
DRAWTOOLS = draw_tools(
  PEN,
  BRUSH,
  DRAWMODE,
  FONT,
  COLOR Foreground,
  COLOR Background,
  BK_MODE)
\end{verbatim}
Normally you would use one or more of the tool-specific predicates to change only the tools that need to be changed.

Predicates for the individual tools are described below.

**Pens**

Lines and the periphery of figures are drawn using a PEN. The way a line will look, i.e. its width, style and color, can be changed by setting the pen attributes before the drawing is done.

The domain for pens is:

GLOBAL DOMAINS

- PENWIDTH = INTEGER
- PEN = pen(PENWIDTH, PENSTYLE, COLOR)

<table>
<thead>
<tr>
<th>Style</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ps_Hollow</em></td>
<td>Selects an invisible pen.</td>
</tr>
<tr>
<td><em>ps.Solid</em></td>
<td>Solid pen.</td>
</tr>
<tr>
<td><em>ps.Dash</em></td>
<td>Dashed pen.</td>
</tr>
<tr>
<td><em>ps.Dot</em></td>
<td>Dotted pen.</td>
</tr>
<tr>
<td><em>ps.DashDot</em></td>
<td>Alternating dashes and dots.</td>
</tr>
<tr>
<td><em>ps.DashDotDot</em></td>
<td>Alternating dashes and double dots.</td>
</tr>
</tbody>
</table>

**Get the Current Pen**

The *win_GetPen* predicate returns the current Pen. This is often used to temporarily change the pen and set it back to the old value after drawing.

\[
\text{Pen} = \text{win\_GetPen}(\text{Win})
\]

**Setting the Pen to be Current for a Window**

The *win_SetPen/2* predicate changes the pen used for drawing in a given window. The following code sets a red pen with the solid style and a width of two pixels.

*Chapter 6, Drawing Operations*
PenWidth = 2,
Pen = pen(PenWidth, ps_Solid, color_Red),
win_SetPen(Win, Pen),

**Background Modes**

When drawing text, the text is drawn with the current foreground color, and if the BK_MODE is set to bk_Opaque, the background will first be filled with the current background color. By using the value bk_Transparent, the text will be drawn directly on whatever is underneath it.

GLOBAL DOMAINS

BK_MODE = INTEGER

CONSTANTS

bk_Transparent = 1
bk_Opaque = 2

To change the current background mode use the following predicate:

win_SetBackMode(Win, BK_MODE)

**Brushes**

Figures are filled using a Brush. The way a brush paints is determined by the brush attributes, which consist of a color and a fill pattern style.

GLOBAL DOMAINS

BRUSH = brush(PATSTYLE, COLOR)

A table of fill pattern styles will be found below. A table of the basic 16 colors is on page 88.

<table>
<thead>
<tr>
<th>Style Identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat_Hollow</td>
<td>Hollow brush</td>
</tr>
<tr>
<td>pat_Solid</td>
<td>A solid brush with specified color</td>
</tr>
<tr>
<td>pat_Horz</td>
<td>Horizontal hatch</td>
</tr>
<tr>
<td>pat_Vert</td>
<td>Vertical hatch</td>
</tr>
</tbody>
</table>

Table Error! Bookmark not defined..10: Brush Fill Pattern Styles
<table>
<thead>
<tr>
<th>pat_Fdiag</th>
<th>45-degree downward hatch (left to right)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat_Bdiag</td>
<td>45-degree upward hatch (left to right)</td>
</tr>
<tr>
<td>pat_Cross</td>
<td>Horizontal and vertical cross-hatch</td>
</tr>
<tr>
<td>pat_DiagCross</td>
<td>45-degree cross-hatch</td>
</tr>
</tbody>
</table>

**Getting and Setting the Current Brush**

The `win_GetBrush` predicate returns the current Brush

```
CurrentBrush = win_GetBrush(Win)
```

The `win_SetBrush/2` predicate changes the current brush for a window.

```
win_SetBrush(Win, NewBrush)
```

**Fonts**

It is possible to change the current font by a call to:

```
win_SetFont(Win, FontBinary)
```

You can retrieve the current font by a call to:

```
FontBinary = win_GetFont(Win)
```

Please refer to the section about font handling (page 84) for an explanation on how to create or select alternate fonts.

**Drawing Modes**

The drawing mode specifies how pixels that are drawn on the screen combine with those already there. The drawing mode options are listed in the table below.

The normal drawing mode is `dm_CopyPen` which just sets the pixels to the new values. `dm_XorPen` or `dm_Not` can be used to draw rubber bands, where the 'stretching' effect is created by drawing a rectangle, and erasing it by drawing a second time (See VPI\EXAMPLES\MAPPING for an example of such code).

**Getting and Setting the Current Drawing Mode**

The `win_GetDrawMode/1` predicate obtains information about the current drawing mode:
DrawMode = win_GetDrawMode(Win)

The **win_SetDrawMode/2** predicate sets a new drawing mode:

```
win_SetDrawMode(Win, dm_Copy)
```

**Table Error! Bookmark not defined..11: Available Drawing Modes**

<table>
<thead>
<tr>
<th>Mode Identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dm_Black</code></td>
<td>Black.</td>
</tr>
<tr>
<td><code>dm_NotMergePen</code></td>
<td>Not(Pen</td>
</tr>
<tr>
<td><code>dm.MaskNotPen</code></td>
<td>(Not(Screen Pixel)) &amp; Pen</td>
</tr>
<tr>
<td><code>dm_NotCopyPen</code></td>
<td>Not(Pen)</td>
</tr>
<tr>
<td><code>dm.MaskPenNot</code></td>
<td>(Not(Pen)) &amp; Screen Pixel</td>
</tr>
<tr>
<td><code>dm_Not</code></td>
<td>Not(Screen)</td>
</tr>
<tr>
<td><code>dm_XorPen</code></td>
<td>Pen Xor Screen Pixel</td>
</tr>
<tr>
<td><code>dm_NotMaskPen</code></td>
<td>Not(Pen &amp; Screen Pixel)</td>
</tr>
<tr>
<td><code>dm.MaskPen</code></td>
<td>Pen &amp; Screen Pixel</td>
</tr>
<tr>
<td><code>dm_NotXorPen</code></td>
<td>Not(Pen Xor Screen Pixel)</td>
</tr>
<tr>
<td><code>dm_Nop</code></td>
<td>Screen not changed</td>
</tr>
<tr>
<td><code>dm_MergeNotPen</code></td>
<td>(Not(Screen Pixel))</td>
</tr>
<tr>
<td><code>dm_CopyPen</code></td>
<td>Pen</td>
</tr>
<tr>
<td><code>dm_MergePenNot</code></td>
<td>(Not(Pen))</td>
</tr>
<tr>
<td><code>dm_MergePen</code></td>
<td>Pen</td>
</tr>
<tr>
<td><code>dm_White</code></td>
<td>White.</td>
</tr>
</tbody>
</table>

**Drawing Predicates**

The following summarizes the available drawing operations. For displaying bitmaps or graphics, please refer to the Pictures section (page **Error! Bookmark not defined.**).
Drawing a Pixel

The `draw_Pixel/3` predicate sets a pixel to a given color.

\[
\text{draw_Pixel}(\text{WINDOW Window}, \text{PNT Position}, \text{COLOR Color})
\]

The reverse operation is performed by:

\[
\text{Color} = \text{Win_GetPixel}(\text{Win}, \text{pnt}(X, Y))
\]

This returns the color of the pixel at logical coordinate \((X, Y)\) in window \textit{Win}.

Filling an Area

\[
\text{draw_FloodFill}(\text{Win}, \text{PNT}, \text{StopColor})
\]

changes the pixel at coordinate \((X, Y)\) to the color specified by the last `win_SetBrush/2` operation for the window. The filling operation is then recursively applied to all neighbor pixels until the color \text{STOP\_COLOR} is reached. If not reached, the drawing window will be completely filled.

Drawing Icons

\[
\text{draw_Icon}(\text{Win}, X, Y, \text{resource_identifier})
\]

This predicate retrieves an icon with the given resource identifier from the resource segment of the .EXE file of the active application and draws it. The icon's top left corner is placed at logical coordinate \((X, Y)\).

Drawing Open Figures

\[
\text{draw_Line}(\text{Win}, \text{PNT}, \text{PNT})
\]

\[
\text{draw_Arc}(\text{Win}, \text{Rct}, \text{INTEGER StartX, INTEGER StartY, INTEGER StopX, INTEGER StopY})
\]

\[
\text{draw_Polyline}(\text{Win}, \text{PntList})
\]

These predicates draw figures using with the Pen without filling by the Brush.

Drawing Filled Figures

\[
\text{draw_Ellipse}(\text{Win}, \text{Rct})
\]
draw_Pie(Win, Rct, INTEGER StartX, INTEGER StartY, INTEGER StopX, INTEGER StopY)
draw_Polygon(Win, PNTLIST)
draw_Rect(Win, Rct)
draw_RoundRect(Win, Rct, INTEGER EllipseWidth, INTEGER EllipseHeight)

The shape around the figures is drawn by the current Pen. These figures are then automatically filled by the current Brush.

**Drawing Text**

There are many options when drawing text:

- draw_Text(Win, X, Y, String)
- draw_Text(Win, X, Y, String, DrawLength)

**draw_Text** draws the first DrawLength characters from String in the window Win, where X and Y are the logical coordinates for the top left corner of the first letter in the string. You can omit DrawLength or set it to -1 to draw the entire string.

- draw_TextInRect(Win, Rct, String, DrawLength, ILIST DrawFlags)

The **draw_TextInRect** function writes the first DrawLength characters from String within a clipping rectangle, using the currently selected font. The dtext_Flag possibilities are shown in the table below:

**Table Error! Bookmark not defined..12: DrawText Flags**

<table>
<thead>
<tr>
<th>Flag Identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtext_center</td>
<td>Centers the text horizontally in the rectangle</td>
</tr>
<tr>
<td>dtext_left</td>
<td>Aligns the text to the left of the rectangle.</td>
</tr>
<tr>
<td>dtext_right</td>
<td>Aligns the text to the right of the rectangle.</td>
</tr>
<tr>
<td>dtext_vcenter</td>
<td>Centers the text vertically. Only works in</td>
</tr>
<tr>
<td></td>
<td>combinations with dtext_singleline.</td>
</tr>
<tr>
<td>dtext_bottom</td>
<td>Specifies that the text is aligned to the bottom</td>
</tr>
<tr>
<td></td>
<td>of the rectangle. Only works with dtext_singleline.</td>
</tr>
</tbody>
</table>
Measuring Text

It is often important to know beforehand what the presentation size of a text string will be. The `win_GetTextExtent/5` predicate returns the height and width (in logical units) of the rectangle that would contain the text were it to be drawn in the given window. The third parameter gives the number of characters to include in the measurement, and setting the value to -1 includes the entire string.

```
Text = "This is a test",
win_GetTextExtent(Win, Text, -1, Width, Height),
```

Coloring Text

There are two predicates to change the foreground and background color for the drawing text:

```
win_SetBackColor(Window, Color)
win_SetForeColor(Window, Color)
```

A table of identifiers for the basic 16 colors is be found on page 88. There are no predicates for directly retrieving the current foreground and background colors, however they may be obtained from the `draw_tools` structure returned by a call to `win_GetDrawTools` for that window.

At `draw_Text` time, if the current background drawing mode is `bk_Opaque` for the `draw_Text` window the rectangle is filled with the color from the last issued `win_SetBackColor/2` predicate for that window, then the text is drawn. If the background-drawing mode for the `draw_Text` window is `bk_Transparent`, the text is drawn on the existing background.
% Draw Text in White on Yellow background
win_SetForeColor(Win, color_White),
win_SetBackColor(Win, color_Yellow),
win_SetBackMode(Win, bk_Opaque),
Text = "This is some text"
X = 20, Y = 20,
draw_Text(Win, X, Y, Text),

Font Handling

Each window has a font, which is used when drawing text. This font can be changed by calling the \texttt{win\_SetFont} predicate.

A font can either be selected by the user in a call to the common font dialog - or it can be created based on font family, font style and font size by a call to \texttt{font\_Create}.

A font is a VPI binary domain. This means that it can be stored in a file or external database to be used at a later time. It is not possible to inspect the settings directly but the predicate \texttt{font\_GetAttrs} can retrieve some information.

Font Creation

\texttt{Font = font\_Create (FontFamily, StyleFlags, Size)}

This returns a font structure. For the font family and for the style flags there are the following possibilities:

\begin{table}
\begin{tabular}{|c|c|}
\hline
Font Family & Description \\
\hline
ff_System & \\
ff_Fixed & \\
ff_Times & \\
ff_Helvetica & \\
\hline
\end{tabular}
\end{table}
Table \textit{Error! Bookmark not defined.}\texttt{.14: Font Style Flags}

<table>
<thead>
<tr>
<th>Font Style Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs_Bold</td>
</tr>
<tr>
<td>fs_Italic</td>
</tr>
<tr>
<td>fs_UnderLine</td>
</tr>
</tbody>
</table>

The \textit{StyleFlags} parameter in font creation is a list so that more than one can be specified. An empty list will create a normal font.

\textbf{Font Selection}

To set a new font in a window use the \texttt{win\_SetFont/2} predicate:

\begin{verbatim}
    winSetFont(Win, Font)
\end{verbatim}

To change the attributes of a font use the predicate:

\begin{verbatim}
    fontSetAttrs(Font, FontStyle, FontSize)
\end{verbatim}

\textbf{Obtaining Font Information}

Size information for the current font can be obtained by using the \texttt{win\_GetFontMetrics/4} predicate.

\begin{verbatim}
    winGetFontMetrics(Win, Leading, Ascent, Descent)
\end{verbatim}

\textit{Win} is the handle to the window, \textit{Leading} is the amount of extra leading (space) that the application adds between rows. \textit{Ascent} is the space between the base line and the top of the character cell. \textit{Descent} is the space between the bottom of the character cell and the base line.

Style information for the current font is obtained by using the \texttt{font\_GetAttrs/3} predicate:

\begin{verbatim}
    FontName = fontGetAttrs(Font, StyleFlags, FontSize),
\end{verbatim}

Here, the returned \textit{StyleFlags} is a list of styles, returned \textit{FontSize} is an integer, and the returned \textit{FontName} is the font name.

It is possible to retrieve the current font in a window by a call to:

\begin{verbatim}
    Font = winGetFont(Win)
\end{verbatim}
Note that the FONT produced by Windows is the best approximation that the GDI can make to the requested characteristics. In some cases it might be of an unexpected size. This size however should not normally exceed the size requested, but that can happen too.
CHAPTER 7

Color Handling

The VPI uses an RGB color model, which allows you to work with 16 million colors. You compose your desired color from values specifying how much Red (0 to 255), how much Green (0 to 255) and how much Blue (0 to 255) you want in your color.

As opposed to coloring by painting, which is termed subtractive because each added color blocks the color below, coloring by the use of light is termed additive because the color of each pixel on a display is specified by adding the intensity values for each of the primary colors: red, green, and blue. For example, if the minimum intensity on a given display was specified as 0 and maximum intensity was specified as 255, a white pixel would be identified by the RGB triplet (255, 255, 255), a black pixel would be identified by the RGB triplet (0, 0, 0), and a cyan pixel would be identified by the RGB triplet (0, 255, 255).

Depending on your video adapter and printer you might actually only be able to display a smaller set of the possible colors. A monochrome printer will produce output in black and white and the video card might produce only 16 colors. However, the underlying native GUI software provides advanced color mappings and color compositions, so if you try to draw 2000 rectangles each with a different color, the underlying GUI can often arrange the colors and pixels in these rectangles to approximate the 2000 different colors. On monochrome devices shades of gray produced by printing of pixel patterns are used to represent colors.

Palette technology is also used to provide additional flexibility in available colors. Color palettes are used by devices that are capable of generating many colors but that can only display or draw a subset of these at any given time. A palette is an array of the colors used in a picture and permits each member color to be specified. When the screen card supports, for example, 256 colors simultaneously, it means, that it can display 256 colors correctly, the palette can specify which 256 colors can be displayed. The windowing system has one global system palette, and each window can have a local palette. The windowing system will try to display the colors in the window that has the focus as correctly as possible. This makes it possible, for example, to set 200 of the 256 possible
colors of a particular video card to various shades of light blue, with the other 56 colors being available for other windows on the screen.

Fortunately, neither the programmer nor the user normally have to worry about color handling. The user knows that the better hardware bought, the better the screen looks. Unless displaying pictures or something else with many colors, the programmer can just choose any color for his GUI elements. However because there are the above mentioned limitations on the total number of colors that can be simultaneously displayed on the screen, and also to provide better portability to different machines, it is a good idea where possible to limit the usage of colors to the constant values defined in VPI.CON:

<table>
<thead>
<tr>
<th>Color:</th>
<th>Value (B-G-R):</th>
</tr>
</thead>
<tbody>
<tr>
<td>color_Red</td>
<td>0x0000FF</td>
</tr>
<tr>
<td>color_Green</td>
<td>0x00FF00</td>
</tr>
<tr>
<td>color_Blue</td>
<td>0xFF0000</td>
</tr>
<tr>
<td>color_Cyan</td>
<td>0xFFFF00</td>
</tr>
<tr>
<td>color_Magenta</td>
<td>0xFF00FF</td>
</tr>
<tr>
<td>color_Yellow</td>
<td>0x00FFFF</td>
</tr>
<tr>
<td>color_Black</td>
<td>0x000000</td>
</tr>
<tr>
<td>color_DkGray</td>
<td>0x404040</td>
</tr>
<tr>
<td>color_Gray</td>
<td>0x808080</td>
</tr>
<tr>
<td>color_LtGray</td>
<td>0xC0C0C0</td>
</tr>
<tr>
<td>color_White</td>
<td>0xFFFFFF</td>
</tr>
</tbody>
</table>

If you want to make your applications look good on monochrome equipment, you should use the color selections color_Black, color_DkGray, color_Gray, color_LtGray, and color_White. There is an attribute attr_have_color that can be used in vpi_GetAttrVal to test if the display hardware supports color. Otherwise make sure that the color contrast is large enough that two adjacent but different colors are not mapped to the same gray color.

The predicate:
COLOR = vpi_ComposeRGB(Red, Green, Blue),
can be used to create a color from the basic red, green and blue primary colors. The luminosity of each primary color is specified as an integer but only the low order 8 bits are significant, giving a range 0 to 255.

The common dialog:

COLOR = dlg_ChooseColor(COLOR OldColor)
can be used to let the user choose any of the possible colors.

Color handling for the various drawing predicates is explained for each predicate and illustrated in the VPI\EXAMPLES\DRAWING example project.

The VPI\EXAMPLES\COLOR example project shows how to draw a number of rectangles with different colors. It shows how to compose and decompose colors, and illustrates that the color value retrieved in win_GetPixel, may not be the same as the color value specified during drawing depending upon the number of available colors on the video adapter card.

When choosing colors for window backgrounds, the application should use the default settings selected in the GUI environment. For this purpose, there are some attributes that can be used to retrieve these color values, as example attr_color_menu return the color for the menus, and attr_color_window will return the color, which should be used for drawing the window background.
CHAPTER 8

Coordinate Systems

In the default logical coordinate system used in the VPI, the X axis goes from the left to the right, and the Y axis from the top to the bottom. By default, the logical units will be screen picture elements (pixels), so that (0,0) would represent the top left corner pixel of a presentation space and lie just immediately inside the top left corner of the bounding rectangle. Since these coordinates are zero-based, any attempt to address the pixel located at coordinates (Width, Height) will be clipped, since it would lie outside the presentation space and in fact on the bounding rectangle itself. It is also possible to work with other logical units and coordinate mapping systems - see the section on Mapping, Scaling, Panning and Zooming (page 92).

When creating, moving or re-sizing a child window, the coordinates are relative to the parent window (Parent Coordinates). In manipulations with task windows and dialogs the coordinates are always relative to the screen (Screen Coordinates).

When drawing or doing any similar operations in a window, the coordinates are relative to the window's client rectangle (Client Coordinates).

Sometimes, however, it is necessary to map coordinates in one window to those of another window, for this purpose there is the predicate win_MapPoints that can convert coordinates which are relative to one window into coordinates relative to another window:

\[
\text{NewPntList} = \text{win\_MapPoints(FromWin, ToWin, OldPntLists)}.\]

For example, if ToWin is the parent window of FromWin window then win_MapPoints recalculates client coordinates of points to parent window coordinates, if ToWin is the Screen window then win_MapPoints converts client coordinates to screen coordinates (and vice versa).

When working with rectangles the values are always rect(Left, Top, Right, Bottom).

Important Note: The VPI does NOT specify rectangles as \((X, Y, Width, Height)\).
Normally coordinates are expressed in pixels, which is the right choice when working with editors, menus, trees, toolbars etc. in windows on the screen. Screen resolution is usually misstated as numbers of pixels in the X and Y directions (e.g. 640 x 480 or 1024 x 768) whereas more properly it should be a measure of pixels per centimeter or inch. The "dot pitch" is the center to center distance of adjacent physical pixel areas, which of course is the inverse of the resolution. Higher resolution screens have greater numbers of pixels, enabling more information to be presented at the same time on the screen. Twice the resolution would permit presentation of 4 times as much information without loss of detail. However, each item displayed becomes a quarter the size, necessitating much closer viewing or a display with four times the area (twice the diagonal measurement of the active area).

**Dialog Base Units**

However, dialog and control sizes normally are multiples of the font chosen for the dialog that, by default, is the system font, which might be changed by the user. That means that their coordinates are scaled in the Dialog Base Units. When the system font changes, the size of text controls will change and the whole dialog will be scaled accordingly.

VPI supports two kinds of Dialog Base Units: *Windows Based* and *Platform Native*.

There is a predicate:

```
vpi_GetBaseUnits(INTEGER Width, INTEGER Height)
```

which returns the pixel sizes of the horizontal (width) and vertical (height) platform native Dialog Base Units. They are calculated as the average width/4 and height/8 of the characters for the system font. These values should be used for positioning and layout of windows and dialogs that depend on the system font.

*win_CreateDyn, win_CreateDynControl* and *win_CreateDynDialog* support the possibility to use both kinds of Dialog Base Units to calculate the sizes and positions of windows, dialogs, and controls. The kind of units must be specified in the object description (WINDEF domain) by the *u_DlgBase, u_DlgPlatform* or *u_Pixels* flag in the *UType* subcomponent of the WDEF sub-structure of the WINDEF domain.

Using of Platform Native Dialog Base Units in these predicates in case of multi-platform projects has the following imperfection. When dialog layouts are designed under one platform but the target application should be used under both
Windows and OS/2 platforms then the width of dialogs and controls under Windows will be about 30% wider then under OS/2.

To avoid this problem the Windows Based Dialog Base Units can be used, in this case, width of dialogs and controls under both Windows and OS/2 platforms will be the same. However, in this case the width of the text strings under Windows will be about 30% wider then under OS/2. Therefore, in this case if you design layouts under OS/2, you must check that text strings satisfy to the controls' width under Windows.

The following convention to convert coordinates scaled in pixel to coordinates scaled in Platform Native Dialog Base Units can be used:

- \[ \text{NatDBU}_X = \left( \frac{X\text{pixel} \times 4}{\text{NatDlgBaseWidth}} \right) \]
- \[ \text{NatDBU}_Y = \left( \frac{Y\text{pixel} \times 8}{\text{NatDlgBaseHeight}} \right) \]

under both Windows and OS/2.

The relations between Windows Based and Platform Native Dialog Base Units are the following:

- under OS/2:
  - \[ \text{WinDlgBaseWidth} = \frac{4}{3}\times\text{NatDlgBaseWidth} \]
  - \[ \text{WinDlgBaseHeight} = \text{NatDlgBaseHeight} \]

- under Windows:
  - \[ \text{WinDlgBaseWidth} = \frac{4}{3}\times\text{NatDlgBaseWidth} \]
  - \[ \text{WinDlgBaseHeight} = \text{NatDlgBaseHeight} \]

**Mapping, Scaling, Panning and Zooming**

These words all cover the same basic operation - mapping. The mapping facility allows the application to change logical coordinate and scaling systems, which are converted and scaled by the underlying GUI system to the actual physical window, printer or Fax coordinates, thus making the application device independent.

Once a mapping mode has been set up for a window, the application must give and will receive logical coordinates instead of the physical coordinates of the actual device. For example if the logical X coordinate ranges from 10000 to 20000 and the logical Y coordinate ranges from 10000 to -10000, then mapping can be set up to map these ranges into the actual physical pixel window coordinates, for example with physical X ranging from 0 to 349 and physical Y from 0 to 249.
**win_SetMapMode (Window, MapMode)**

A call to the predicate `win_SetMapMode` will change a window's mapping mode. The possible mapping modes are listed in the table below.

The default setting is always `mm_Text`, so that the coordinates correspond to the actual pixels on the screen or printer. The `mm_Text` is device dependent mode while the other modes are device independent. The *Metric*, *English* and *Twips* mapping modes allow the application to work with actual physical sizes, which is especially useful for printing. The mode `mm_Arbitrary` allows any range of logical coordinates to be mapped to the actual range of coordinates in a window.

### Table Error! Bookmark not defined. 16: Mapping Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mapping</th>
<th>X+Direction</th>
<th>Y+Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mm_Text</code></td>
<td>Logical = Physical (default).</td>
<td>Right</td>
<td>Down</td>
</tr>
<tr>
<td><code>mm_Arbitrary</code></td>
<td>Set by <code>win_SetMapScale()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mm_HiMetric</code></td>
<td>Logical = 0.01 millimeter.</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td><code>mm_LoMetric</code></td>
<td>Logical = 0.1 millimeter.</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td><code>mm_HiEnglish</code></td>
<td>Logical = 0.001 inch.</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td><code>mm_LoEnglish</code></td>
<td>Logical = 0.01 inch.</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td><code>mm_Twips</code></td>
<td>Logical = 1/20 of a point. (72 points = 1 inch)</td>
<td>Right</td>
<td>Up</td>
</tr>
</tbody>
</table>

**win_SetMapScale(Win, PNT DevOrg, PNT DevExt, PNT LogOrg, PNT LogExt)**

**win_GetMapScale(Win, PNT DevOrg, PNT DevExt, PNT LogOrg, PNT LogExt)**

The predicate `win_SetMapScale` can be used to set up any scaling in logical coordinates.
When the mapping mode is `mm_Arbitrary`, the following coordinate transformations will be done for both the X and the Y coordinate:

\[
\text{DevCoord} = \text{DevOrg} + (\text{LogCoord} - \text{LogOrg}) \times \frac{\text{DevExt}}{\text{LogExt}}
\]

where \text{LogCoord} is the coordinate in logical units and \text{DevCoord} is the resulting in coordinate physical units.

By changing only the \text{DevOrg} argument, it is possible implement panning (moving the origin of the coordinate system). By changing the \text{DevExt} and \text{LogExt} it is possible to scale and zoom a drawing.

Using the same scaling on each axis gives Isotropic scaling, and using different scaling gives Anisotropic scaling.

The following predicate can be used to retrieve the current mapping scale of that window:

```
win_GetMapScale(Win,
    PNT DevOrg,
    PNT DevExt,
    PNT LogOrg,
    PNT LogExt)
```

\text{LogPntList} = \text{win\_DPtrtoLP}(\text{WIN}, \text{PNTLIST DevicePoints})

\text{DevPntList} = \text{win\_LPtoDP}(\text{WIN}, \text{PNTLIST LogPoints})

This two predicates can be used to convert device coordinates (screen pixels, printer or fax dots) to logical coordinates and back. This is useful because the values in the Mouse and Update events etc. are always given in pixel coordinates.

Unfortunately, on 16-bit Windows the scale for logical coordinates is limited to \(-32000 < X < +32000\) because the coordinates are handled as 16-bit integers.

### Mapping Examples

#### Drawing a Map

In the VPI\EXAMPLES\MAPPING directory there is an example showing how to draw a map of Denmark, which supports zooming in and out on this map. There is a fact database of polygons representing the coastline in a coordinate system from \(-10000\) to \(+10000\). Initially this coordinate system is mapped to the actual window device coordinates. When zooming is done, the act of the user
dragging a rectangle sets up a mapping for zooming the selected area to fill the window.

**Printing**

When creating print routines for a window on the screen, there is often the problem that the window might consist of several Child windows, which need to be printed separately by different routines. One easy solution is to use mapping to properly position each of the Child windows on the page. The mapping can also scale everything back up again to natural size because of the higher resolution of the printer. See the example VPI\EXAMPLES\PRINT for how to use mapping to scale faxed or printed output.
CHAPTER 9

Printing

Printing is very similar to drawing in a window. During printing a special Printer Window handle is used in all the drawing functions. Printing is handled one page at a time and drawing may occur at any place on the page in any sequence. There is a standard VPI dialog `dlg_PrintSetup` (calling the Print dialog box) which enables operator selection of the range of pages to print (but that selection is not made available to the application), the printer resolution, number of copies and whether they should be collated, selection of printer to use (but not to change the selection of default printer), the orientation of printing on the page (portrait or landscape), paper size and source bin, dithering choice, print intensity slider and whether to print TrueType fonts as graphics. All these will then apply for the whole print job.

For MS Windows, the default printer may only be changed by native Windows applications or through the Windows Control Panel, but this will not change the printer in any VPI applications that are already running using the default printer. VP printing may be performed to a fax driver.

Today's windowing systems do an impressive job of letting applications be device independent. The windowing system knows everything about handling the specifics of individual printers, how to set bold or italic typefaces, how to do page shifts etc. For the application there is generally no programming difference between drawing on the screen, a laser printer, a plotter or a daisy-wheel or typewriter-style printer, or a facsimile device. The main difference between the screen and the printer is usually that there is much higher resolution on the printer, and that the printer can only print in black and white. If nothing special is done to scale up the print output, it may appear very small on the printer.

If an application is to print WYSIWYG output, it should scale the printing based on the printer and display resolutions. This information is returned by calling `vpi_GetAttrVal` (see page 131). Arguments provide the ability to obtain the number of Dots Per Inch on the screen and on the printer. Scaling by these values makes the drawing on paper appear similar to the drawings on the screen. The following framework can be used to scale printing done in pixel coordinates:
VResPrinter = vpi_GetAttrVal(attr_printer_vres),
HResPrinter = vpi_GetAttrVal(attr_printer_hres),
VResScreen = vpi_GetAttrVal(attr_screen_vres),
HResScreen = vpi_GetAttrVal(attr_screen_hres),
PRINTWIN = print_StartJob("Test printing. . . "),
win_SetMapMode(PRINTWIN, mm_Arbitrary),
win_SetMapScale(PRINTWIN,
    pnt(0, 0), pnt(VResPrinter, HResPrinter),
    pnt(0, 0), pnt(VResScreen, HResScreen)),
print_StartPage(PRINTWIN),
: draw_xxx(PRINTWIN), <------------- The actual drawing of a page
: print_EndPage(PRINTWIN),
print_EndJob(PRINTWIN).

See VPI\EXAMPLES\PRINT for how to handle the difference between drawing in a window and to the printer, and VPI\EXAMPLES\PRINTTXT for an example of how to print text files.

The VPI contains six predicates to support printing:

**Start a Print Job**

WINDOW = print_StartJob(Title)

To start a new print use `print_StartJob`. This will return a Window handle that can be used as the target object of drawing routines and the other printing predicates.

**End a Print Job**

print_EndJob(PrintWin)

When all printing has been completed a call to `print_EndJob` must be done.

**Set a Page Orientation**

print_SetOrientation(IsOrientationLandscape)

Before starting a new printing job the pages orientation can be changed (under MS Windows underlying platforms) by a call to `print_SetOrientation`. The
default orientation is Portrait. Specify $IsOrientationLandscape = b_{\text{true}}$ to set Landscape orientation.

**Start a New Page**

\[
\text{print\textunderscore StartPage(WINDOW PrintWin)}
\]

When printing, the application must print one page at a time. Before starting to print a page the `print\textunderscore StartPage` predicate must be called. For MS Windows, if this is called and the printer supports banding, the system will automatically perform the banding.

**End a Printed Page**

\[
\text{print\textunderscore EndPage(WINDOW PrintWin)}
\]

After all output to the page has been completed the predicate `print\textunderscore EndPage` is called.

**Abort a Print Job**

\[
\text{print\textunderscore AbortJob(PrintWin)}
\]

If the application for some reason wants to stop the printing, the predicate `print\textunderscore AbortJob` can be called.

While printing a file the dialog "File printing" is displayed. The user can interrupt printing by pressing the "Cancel" button. This dialog contains two strings with the default texts: "Page" and "Cancel". These strings have the fixed resource identifiers: `vpi\_strcon\_prn\_page` and `vpi\_strcon\_prn\_cancel` respectively. Therefore the programmer can easily edit these strings using the Resource Strings Editor.

**Retrieve Printing Settings**

Calling `print\textunderscore GetConfig` predicate:

\[
\text{PrintConfig = print\textunderscore GetConfig()}
\]

it is possible to retrieve information about some of the user's selections done in the Print dialog box. The returned printer configuration record `PrintConfig`
contains information that the `dlg_printSetup` predicate uses to initialize the Print dialog box and `PrintConfig` is intended only for further use in a call:

```plaintext
print_SetConfig(PrintConfig)
```

to restore printing settings. You can save `PrintConfig` to a file and read it back, but remember that the contents of the printer configuration record depend on the underlying windows platform and do not platform compatible.
CHAPTER 10

Printing

The Cursor is the pointing shape for the Mouse or track ball device. Each window on the screen has its own cursor, which can be changed by the application. The cursors that can be set to a window are either predefined, or they can be specified in the resource file.

There are 11 predefined cursor types available. These predefined constants belong to the CURSOR domain and are defined in the table below.

Additional cursors can be defined in the resource file.

If you need user defined cursors they must be given a resource id (constant) of 12 or more. The VDE supplies a constant value automatically when you register a cursor by using the Cursor and New buttons of the project (.VPR) window. The Cursors in the project can either be created by the VP image editor, or any external tool that can store the cursor data in a separate .CUR file.

<table>
<thead>
<tr>
<th>Cursor Identifier</th>
<th>Description of Cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursor_Arrow</td>
<td>Standard arrow cursor.</td>
</tr>
<tr>
<td>cursor_Cross</td>
<td>Crosshair cursor.</td>
</tr>
<tr>
<td>cursor_Ibeam</td>
<td>Text I-beam cursor.</td>
</tr>
<tr>
<td>cursor Icon</td>
<td>Empty icon.</td>
</tr>
<tr>
<td>cursor Size</td>
<td>A square with a smaller square inside its lower right hand corner.</td>
</tr>
<tr>
<td>cursor SizeNESW</td>
<td>Double-pointed cursor pointing northeast/southwest.</td>
</tr>
<tr>
<td>cursor SizeNS</td>
<td>Double-pointed cursor pointing north/south.</td>
</tr>
<tr>
<td>cursor SizeNWSE</td>
<td>Double-pointed cursor pointing northwest/southeast.</td>
</tr>
<tr>
<td>cursor SizeWE</td>
<td>Double-pointed cursor pointing east/west</td>
</tr>
<tr>
<td>cursor_Uparrow</td>
<td>Vertical arrow cursor</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>cursor_Wait</td>
<td>Hourglass cursor</td>
</tr>
<tr>
<td>cursor_User</td>
<td>First resource ID to use for your own designs</td>
</tr>
</tbody>
</table>

**Cursor Predicates**

The VPI has four predicates for handling cursors, they are described in the VPI help file.

`cursor_Set(Window, CursorID)`

This immediately replaces the cursor of the window referred to by `Window` with the one defined by `CursorID`.

`cursor_Set(Window, CursorID, ChangeNow)`

On the contrary to the previous, the `cursor_Set(_, _, b_false)` changes cursor only after a mouse movement in the specified window or when the mouse comes into the specified `Window`. On the other hand, the `cursor_Set(_, _, b_true)` works exactly like `cursor_Set/2`.

`cursor_Get(Window)`

This retrieves the current cursor. This is useful if you want to restore the previous cursor after replacing it with the Hourglass cursor.

`cursor_Hide()`

This makes the cursor disappear until the `cursor_Set` predicate is called or the mouse is moved.

`cursor_SetWait()`

This sets the current cursor to Hourglass. Notice that any mouse movement restores the default cursor.

`CursorPosition = cursor_GetPos(Window)`

This retrieves the current cursor (mouse) position related to the given window `Window`. 
Examples

See the example program in the VPI\EXAMPLES\CURSOR directory.
Mouse Handling

There are several different kinds of mouse devices. If a mouse device has only one button (as on a Macintosh) it normally acts as a left mouse button. Some PC mice have three buttons (left, center and right). One button mice also appear in the form of digitizers and touch sensitive screens. Most mouse devices have two buttons, and a center button is simulated when both buttons are pressed down simultaneously.

When a user clicks (or double-clicks) in a window, that window will usually gain the focus. When a window (or control) has the focus it will receive mouse events, which occur within its border.

Capturing the Mouse

With the mouse capture set to a window, all mouse input is directed to that window, regardless of whether or not the cursor is over that window. Only one window at a time can capture the mouse. This can be used as in the VP text editor to automatically scroll the text when the user wants to drag a piece of text to another location.

\texttt{win\_CaptureMouse(Window)}

This predicate redirects all mouse events to the window given by \texttt{Window}. All \texttt{e\_MouseMove} events are still sent to the Window while capturing.

\texttt{win\_ReleaseMouse()}

Releases the mouse after a call to \texttt{win\_CaptureMouse}.

Mouse Events

The VPI translates the native mouse events into the following four operating system independent messages.
e_MouseDown(PNT, Shift_Ctl_Alt_STATUS, Button)
e_MouseUp(PNT, Shift_Ctl_Alt_STATUS, Button)
e_MouseDbl(PNT, Shift_Ctl_Alt_STATUS, Button)
e_MouseMove(PNT, Shift_Ctl_Alt_STATUS, ButtonsList)

Each message has three parameters called PNT, Shift_Ctl_Alt_STATUS, and Button or ButtonsList.

PNT is a structure of the form \(\text{pnt}(x, y)\) where \(x\) and \(y\) are integers that give the coordinates of the mouse, relative to the window's top left hand corner \((0, 0)\).

Shift_Ctl_Alt_STATUS shows the status of the Shift, Control and Alt keys.

Button and ButtonsList are an integer(s) that tells which button(s) has been pressed. The following constants are defined in VPI.CON:

- \text{mouse\_button\_left}\quad\text{Left mouse button pressed / released}
- \text{mouse\_button\_right}\quad\text{Right mouse button pressed / released}
- \text{mouse\_button\_middle}\quad\text{Center button pressed / released}
Rectangles

Rectangles are used in many places in the VPI. Position and size of a window is specified by a rectangle, the windowing system updates the windows in rectangles, it is very common to draw rectangles etc. Because rectangles are used so often, there are some special predicates that implement some common rectangle handling functions:

**Intersection of Two Rectangles**

Rct = rect_Intersect(Rct1, Rct2)

The IntersectRect function calculates the intersection of two rectangles and places the coordinates of the intersection rectangle into the returned rectangle. If the rectangles do not intersect, an empty rectangle rct(0, 0, 0, 0) is returned.

**Inflate the Size of a Rectangle**

Rct = rect_Inflate(Rct, Integer Dh, Integer Dv)

The `rect_Inflate` function increases or decreases the width and height of a rectangle. The `rect_Inflate` function adds `Dh` div 2 units to the left and right ends of the rectangle and adds `Dv` div 2 units to the top and bottom. The `Dh` and `Dv` parameters are signed values; positive values increase the width and height, and negative values decrease them.

**Determine Whether the Area of a Rectangle is Zero**

`rect_IsEmpty(Rct)`

This predicate tests whether a rectangle is empty, which means that the area of the rectangle is zero. The predicate fails if the area of the rectangle is not zero.
Move a Rectangle

\[ \text{RCT} = \text{rect\_Offset}(\text{Rct}, \text{Integer Dh, Integer Dv}) \]

The \text{rect\_Offset} function moves the given rectangle by the specified offsets. \text{Dh} and \text{Dv} are signed values so the rectangle can be moved in any direction.

Determine Whether a Point is Inside a Rectangle

\[ \text{rect\_PntInside}(\text{Rct, PNT}) \]

The \text{rect\_PntInside} predicate determines whether the specified point lies within a given rectangle. A point is within a rectangle if it lies on the left or top side or is within all four sides. A point on the right or bottom side is considered outside the rectangle. The predicate will fail if the point is not inside the rectangle.

Create the Union of Two Rectangles

\[ \text{RCT} = \text{rect\_Union}(\text{RCT Rct1, RCT Rct2}) \]

The \text{rect\_Union} function creates the union of two rectangles. The union is the smallest rectangle that contains both rectangles.
CHAPTER 13

Event Handling

Process Pending Events

\[ \textit{vpi\_ProcessEvents(SendEventsToThisAppBoolean)} \]

This predicate will process the events that are currently in the event queue before it returns. It yields control to other applications before processing of every event from the application's event queue. This is useful in a number of situations.

MS Windows 3.x does not have preemptive scheduling. This means that, when an event-handler predicate is called, no other applications or windows will receive any events before the predicate returns. A similar problem exists in the presentation manager in OS/2 3.x because PM itself runs in a single thread so no other applications will receive events before there is a return from the previous event (timer events are an exception).

This means that it is not desirable to have very long uninterrupted operations like compilation, database report generation, drawing of large pictures, heavy calculations etc. without some mechanism for keeping events flowing to other event-handlers. This is why you must call \textit{vpi\_ProcessEvents} regularly inside such long operations.

As an example the Prolog compiler in the VDE regularly calls \textit{vpi\_ProcessEvents} to support compiling in the background, and so it is possible to keep working in the environment and be running other applications while the compiler is running.

The Boolean parameter \textit{SendEventsToThisAppBoolean} specifies whether \textit{vpi\_ProcessEvents} should cause a processing of events from the application's event queue (b_True) - or it only yields control to other applications in the Windowing environment (b_False). In the second case, it causes processing of only \textit{e\_Timer} and \textit{e\_Update} events are currently in the application's event queue; all other events, except for \textit{e\_Timer} and \textit{e\_Update}, will be just deleted from the application's event queue.
The `vpi_ProcessEvents/0` version of the predicate works exactly like `vpi_ProcessEvents(b_true)`.

```
vpi_ProcessEvents(SendEventsToThisAppBoolean, WaitNewEventsBoolean)
```

The distinction of the `vpi_ProcessEvents/2` from the `vpi_ProcessEvents/1` has place only when the `WaitNewEvents = b_true`. In this case the `vpi_ProcessEvents/2` suspends the application if there is no events in the application's events queue and waits until a new event will come, then it causes a processing of this event.

The `vpi_ProcessEvents/2` is usually used in application's loops in which the application waits for events from other applications.

Note that you should make sure that you can terminate a long process if the user uses `File | Exit` to terminate the application. Instead of calling `vpi_ProcessEvents` you should call a predicate that checks a flag to see if it should terminate before calling `vpi_ProcessEvents`.

### Sending and Posting Events

While events normally originate in the windowing system, your application code can send an event to one of its own event-handlers using the predicate `win_SendEvent`, the effect being to call the event-handler for that window. Of course it is possible to call the event-handler directly if it is accessible at that point, but in many situations only the window handle is known so it is better practice to use a call based on the window handle. The LONG, which is returned by `win_SendEvent`, is the return value (the final argument) of the event-handler clause for that event.

```
LONG = win_SendEvent(Win, EVENT)
```

One use of this would be to relay menu events to the parent window by the call:

```
win_SendEvent(ParentWin, e_Menu(ID, CtrlAltShift)),
```

The call to `win_SendEvent` will not return until the event-handler target clause completes. If the target clause fails, `win_SendEvent` will return zero.

Sometimes it is desirable to put an event into the event queue, so it will be processed after the current event and any other pending events have terminated. `win_PostEvent` will do this:

```
win_PostEvent(WINDOW, EVENT)
```
This predicate returns immediately after placing the event in the queue.
CHAPTER 14

Resources

The term Resources applies to many things in the computing world. In GUI system environments the term resources refers to all the windows, dialogs, pictures, string tables, icons, cursors, device contexts, pens, brushes, memory allocations, etc. for all active applications.

When we refer to resources in Visual Prolog, we normally mean the descriptions of dialogs, menus, tool and help bars, cursors, bitmaps, icons, and string tables that are created by the various resource editors in the VDE. The layout and the attributes of these resources are stored in the project database (<project>.VPR), and when you request a build, the VDE will generate an industry-standard binary resource file (<project>.RES) and a file of Prolog-format constant definitions for inclusion in source modules. The VP code-generators for windows (as opposed to dialogs), and for toolbars and help bars generate Prolog code directly in the application source since such things were not anticipated by the standard .RC or .RES file formats.

Alternatively, or in addition, you may request under Options | Project | Code Generator the generation of <project>.RC and .H files, which provide industry-standard text descriptions of these same resources. These .RC and .H files may be converted into a .RES file by any of the commercially available tools (an RC compiler or any of the resource preparation tools).

Resource Naming

When a resource is created by one of the VDE resource editors, it needs to be symbolically named. The VDE supplies a possible symbol based on a set of rules, and the user can accept, change, or totally replace it. These symbols, which will later become Prolog constants, will be used by the application to refer to resource items, and at compile time each resolves to a unique integer value.

During project building, both a .RES file, destined for the linker, and a .CON file, destined for inclusion in those project source modules concerned with handling resources, are generated (but there are other options as well). The .CON file is the record of the VDE's assignment of a unique integer value to each of the resource item symbols. These numeric values, and not the symbolic names
written in the application, become at run time the actual identification numbers of the resources to the VPI predicates and to the underlying GUI.

The following things go into the resource file: Dialogs, Menus, Cursors, Bitmaps, Icons and Strings. They are all used in different ways.

**Resource Linking**

The .RES file is linked into the executable file (<project>.EXE), in which it forms the resource segment(s). This operation can be done the PDC Linker. During execution, VPI predicates retrieve dialogs, menus, icons, bitmaps, and string table strings from the resource segment of the .EXE.

**Bitmaps, Icons and Cursors**

These can be created either in the VP image editor, or using any third-party tools specifically designed to create, capture or modify graphic images. Unlike dialogs, menus and strings, the definitions of icons, bitmaps and cursors will exist in separate files before they are transferred to the .RES file.

**Dialog Resources**

Dialogs are normally created in the Dialog Editor in the VDE. It removes all the tedious work of positioning and sizing the controls in a dialog. When a definition exists in the resource file, a dialog can be created directly by a call to `win_CreateResDialog` - or the definition can be retrieved into a VPI structure by a call to the predicate:

```
win_GetResDialog(ResID)
```

The following predicate call can be used to check if a dialog definition exists in the resource file:

```
Result = vpi_CheckExistDialog(ResID)
```

**Menu Resources**

Menus are normally designed in the VP menu editor. When windows are created by a call to `win_Create`, the menu for a window is given as a parameter, and by giving the alternative:

```
res_menu(ResId),
```
the window will automatically retrieve the menu from the resource data associated with \textit{ResId}.

**String Resources**

Strings are put into string table resources for several reasons. Having them in resources makes it easy to make changes like translating to another language without having to modify the source code. The application can retrieve the string from a resource identifier by a call:

\[
\text{Str} = \text{vpi\_GetResStr}(\text{ResID})
\]

The following predicate call can be used to check if a string with the specified resource identifier exists in the resource file:

\[
\text{Result} = \text{vpi\_CheckExistString}(\text{ResID})
\]
Menus

The easiest way to give your users a way to select the next action is to organize the possible actions as menu items. The Menu Editor in the VDE makes this task very easy. For each menu item a resource identifier is given to identify the menu entry to the application. Your application can be made more readable by choosing the name of the related action for each resource identifier.

Menus are initially connected to windows during the creation of the window. The menu parameter in the `win_Create` call can specify that the menu is located in the resource file, or is to be built dynamically from a Prolog structure.

When using MDI mode, all menus will appear in the Task window. The menu will automatically change as the active window changes.

For every window to carry its own menu, the MDI flag must not be set in the Application Expert, and the windows must be created with the Screen window as the parent window.

The VPI contains the facility to have owner draw menus. This will not be portable to all platforms. Note also that the menu editor in the VDE only supports menus with all items being text.

**MENU Domains**

```prolog
MENU = res_menu(RESID); dyn_menu(MENU_ITEM_LIST); no_menu

MENU_ITEM_LIST = MENU_ITEM*
MENU_ITEM = ...
```
The **ItemState** identifies the state of a menu item:

- **mis_None** specifies the ordinary state
- **mis_Checked** sets the check mark to the menu item
- **mis_Help** specifies that this menu item must be positioned immediately after the 'Window' menu item; usually, it is used to put the 'Help' item to the far right (only under MDI mode).

Use the following sum:

\[
ItemState = mis_{\text{Checked}} + mis_{\text{Help}},
\]

to specify that the menu item should be positioned immediately after the 'Window' item and it should have the check mark.

**Specifying Actions for Menu Items**

To specify an action for a menu item you should use the Code Expert to generate a clause for that menu event in the appropriate window's event-handler. Any
code placed in the event-handler clause will be invoked when the menu item is chosen (while the menu item is grayed (disabled) your code can never be called - unless there is a tool bar push button which is not disabled and has the same resource identifier).

The following code reacts to the event that is generated when the File | Open item is selected in the Task window menu:

```prolog
    task_win_event_handler(Win, e_Menu(id_file_open, ShiftCtlAlt), 0):-
        dlg_note("Menu", "You chose Open"), !.
```

**Loading a Menu from the Resource File**

The `menu_GetRes` predicate is used to retrieve a menu from a resource file. The menu is loaded into a VPI data structure from the MENU domain.

```prolog
    NewMenu = menu_GetRes(irm_my_menu),
    menu_Set(Win, NewMenu),
```

**Changing the Text for a Menu Item**

The text of an individual menu item can be changed by using the `menu_SetText/3` predicate. The following line inserted into the Task window create event (e_Create) will change the standard File Open text to something different:

```prolog
    menu_SetText(Win, id_file_open, "Open DataBase")
```

**Check Marks**

Menu items can have an associated check mark, which indicates whether some facility is presently available or not. The initial state is given in the definition of the menu, and it is possible to change this state for a menu item in a window by call to the `menu_Check/3` predicate:

```prolog
    menu_Check(Win, menu_item_resource_identifier, BOOLEAN)
```

**Enabling/Disabling Menu Items**

By default new menu items are enabled in the menu editor (except for the ones automatically generated by the Application Expert without code), but they can be
disabled (grayed) or enabled at design by double-clicking the item name (while in the menu editor) or changing the state of the disabled box or at run time by using the `menu_Enable` predicate with the last parameter set to `b_False`.

% Disable the File|New menu item
  menu_Enable(Window, id_file_new, b_False),

% Enable the File|New menu item
  menu_Enable(Window, id_file_new, b_True).

Forcing a Menu Update

If you dynamically change the text of menu items in events other than `e_Create`, the effect of your change may not be visible to the user on all platforms. The `menu_Update/1` predicate invalidates the screen area used by the menu, causing it to be re-drawn.

  menu_SetText(Win, id_file_open, "Open DB"),
% Force Screen Menu Update
  menu_Update(Win)

Handling Sub-Menus under MS Windows

Under MS-Windows, sub-menus cannot be referred to by a resource identifier. If the menu entry for a sub-menu needs to enabled, disabled, checked or have the text changed, it is necessary to use a special versions of the `menu_Enable`, `menu_Check` and `menu_SetText` predicates, which specify the text of the menu entry instead of the constant.

  menu_Enable(Win, String, BOOLEAN)
  menu_Check(Win, String, BOOLEAN)
  menu_SetText(Win, String, NewString)

Pop-Up Menus

The `menu_PopUp/4` predicate is used to create a pop-up menu and would usually be called in response to a mouse click. The following clause:

  my_event_handler(Win, e_MouseDbl(PNT, _, _), 0):- !,
      menu_PopUp(Win, res_menu(idr_my_pop_up), PNT, align_Left).
will pop-up the menu whose resource identifier is \texttt{idr\_my\_popup} at position \texttt{PNT} in window \textit{Win} when the user double clicks inside it.

The last parameter, which can be \texttt{align\_Left}, \texttt{align\_Right} or \texttt{align\_Center}, specifies how the menu will pop-up relative to the specified \texttt{PNT}.

When the user clicks in a pop-up menu for a given window, the VPI sends the usual \texttt{e\_Menu} event to that window's event-handler.
Timers

A timer is an internal routine that repeatedly every time a given interval has elapsed sends an `e_Timer` event to a window event-handler associated with the timer. Because the accuracy of a timer depends on the system clock rate and how often the application retrieves messages from the message queue, the time-out value is only approximate.

A new timer starts timing as soon as it is created. An application can destroy a timer by using the `timer_Kill` function. To use system resources efficiently, applications should destroy timers that are no longer necessary. Each timer has a unique timer identifier, which is returned when the timer is created.

Starting a Timer

A new timer is created with a call to:

```
TimerId = timer_Set(Window, UNSIGNED Interval)
```

The interval is specified in milliseconds. The returned timer identifier can be used to stop the timer.

Stopping a Timer

A timer is destroyed by a call to:

```
timer_Kill(LONG TimerIdentifier)
```

Where the `TimerIdentifier` is the value returned by `timer_Set`. 
CHAPTER 17

Pictures

The VPI contains a group of predicates to handle pictures. Pictures are stored as Windows .BMP files and are not limited to 64 Kb in size. Operations on pictures are done using a handle to the picture, which means once it is finished with, this handle has to be explicitly destroyed to release its allocated resources.

There are four ways to obtain a picture:

Converting Pictures to Binary Data

For storing pictures in databases, sending them over a network etc. it is handy to convert a picture to a standard Prolog binary term. There are two predicates that can do this conversion:

\[
\text{Binary} = \text{pict\_ToBin(PICTURE)} \\
\text{Picture} = \text{pict\_FromBin(BINARY)}
\]

Note, that on 16-bit platforms, there is a 64 Kb - limit on the size of Binary structures, calling \text{pict\_ToBin} on a picture larger than this will result in a run time error.

Creating a Picture from Drawing Operations

It is also possible to create a picture in memory by collecting a number of drawing operations. This requires three steps:

1? Make a call to \text{pict\_Open} to start the collection of drawings.

2? Do all the drawing operations.

3? Make a call to \text{pict\_Close} that returns a handle to the created picture.

\[
\text{PictWin} = \text{pict\_Open(Win, Rct)} \\
: \\
\%\text{ drawing operations into the PictWin} \\
: \\
\text{Picture} = \text{pict\_Close(PictWin)}
\]
Destroying a Picture

When a picture is no longer needed, the allocated memory should be released by a call to `pict_Destroy`:

\[ \text{pict\_Destroy(Picture)} \]

Drawing a Picture

There are two ways to draw a picture, with or without stretching.

No Stretch

A full picture can be drawn at a given location in a window. The top left corner of where the picture is to be drawn relative to the window's client area must be specified in `PNT`.

\[ \text{pict\_Draw(Win, Picture, PNT, ROP)} \]

Stretching

In the stretching version of `pict\_Draw`, the picture will be zoomed to fill the chosen area (DestRct). It is also possible to specify that only part of the picture should be drawn (PictRct), with PictRct referring to the unstretched picture:

\[ \text{pict\_Draw(Win, Picture, DestRct, PictRct, ROP)} \]

The Raster Operations

The ROP argument in `pict\_Draw` specifies an advanced feature indicating how to combine:

1? The existing contents of the screen (Dst)
2? The current brush pattern for the window (Pat)
3? and the picture to be drawn (Src)

The combinations are done bit-wise by logical operations. This feature allows you to make a bitmap, which may be partly transparent, to only update parts of the window, to undraw what was previously drawn, etc.
Table Error! Bookmark not defined. 18: Raster Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>rop_Blackness</td>
<td>Black</td>
</tr>
<tr>
<td>rop_Whiteness</td>
<td>White</td>
</tr>
<tr>
<td>rop_DstInvert</td>
<td>Not $Dst$</td>
</tr>
<tr>
<td>rop_MergeCopy</td>
<td>$Pat$ And $Src$</td>
</tr>
<tr>
<td>rop_MergePaint</td>
<td>(Not $Src$) Or $Dst$</td>
</tr>
<tr>
<td>rop_NotSrcCopy</td>
<td>Not $Src$</td>
</tr>
<tr>
<td>rop_NotSrcErase</td>
<td>Not($Src$ Or $Dst$)</td>
</tr>
<tr>
<td>rop_PatCopy</td>
<td>$Pat$</td>
</tr>
<tr>
<td>rop_PatInvert</td>
<td>$Dst$ Xor $Pat$</td>
</tr>
<tr>
<td>rop_PatPaint</td>
<td>(Not $Src$) Or $Pat$ Or $Dst$</td>
</tr>
<tr>
<td>rop_SrcAnd</td>
<td>Src And $Dst$</td>
</tr>
<tr>
<td>rop_SrcCopy</td>
<td>Src</td>
</tr>
<tr>
<td>rop_SrcErase</td>
<td>(Not $Dst$) And $Src$</td>
</tr>
<tr>
<td>rop_SrcInvert</td>
<td>Src Xor $Dst$</td>
</tr>
<tr>
<td>rop_SrcPaint</td>
<td>Src Or $Dst$</td>
</tr>
</tbody>
</table>

**Get a Picture from the Contents of a Window**

It is possible to retrieve the contents of all or part of the client rectangle of a window by a call to:

```pascal
Picture = pict_GetFromWin(Window, Rct)
```

where $Rct$ specifies the rectangle within the Window to be captured.

**Load a Picture from a File**

Pictures are normally stored with the .BMP (bitmap) extension. A picture can be loaded from a file that stores such a windows bitmap and may be larger than 64 Kb.
Picture = pict_Load(FileName)

**Load a Picture from the Resource Segment**

It is possible to load a bitmap from the resource segment of the application EXE file directly into a picture by a call to:

```
Picture = pict_GetFromRes(resource_identifier)
```

**Returning Size Information from a Picture**

The `pict_GetSize` predicate can be used before drawing to return *Height* and *Width* of a picture - or it can be used to retrieve the size of a picture before converting it to a binary or storing it in a file:

```
pict_GetSize(Picture, INTEGER Width, INTEGER Height, LONG Size)
```

**Saving a Picture to a File**

It is possible to save the contents of a picture in a file by a call to:

```
pict_Save(Picture, FileName)
```

**Rotating a Picture**

It is possible to rotate a picture by a call to:

```
NewPicture=pict_Rotate(Picture, Angle)
```
The Clipboard

The clipboard is the common place where running applications can exchange data through copy and paste operations. Remember that the clipboard is user-controlled; the application should not place anything on the clipboard or clear it without the user's knowledge.

The clipboard data can be in a number of different formats. The VPI supports three portable formats, which it handles in a way compatible with other applications. The three open formats are:

- text, which is correspondent to STRING domain;
- pictures, which are stored as bitmaps;
- metafiles.

You can also put proprietary data formats on the clipboard, which can be exchanged between Prolog applications. These proprietary data are put on the clipboard as a binary combined with a name for the format. Since arbitrary Prolog terms can be converted to binaries, any domain can in principle be stored on the clipboard (subject to the platform size limits of binary data).

There are three things you can do with the clipboard:

1. Test to see if it contains some data,
2. Get the data from the clipboard, and
3. Put some data on the clipboard, which replaces its previous content.

Note: Take a look at the example in the VPI\EXAMPLES\CLIPBRD directory, and see how the VPI formats are compatible with your other programs.

Getting Data from the Clipboard

To retrieve the data from the clipboard, the following predicates can be used. If the selected type of data is not present, the call will fail.
Binary = cb_GetBin(STRING)
Picture = cb_GetPicture()
MetaFileHandle = cb_GetMetafile()
StringLength = cbGetSize()
String = cb_GetString()

Note: the string returned from *cb_GetString* will return a string according to the
Windows conventions about a carriage return–linefeed (CR-LF) combination.
The predicate *str_DOSStr* can be used to convert these CR-LF combinations to a
regular Prolog string with just \n characters.

**Putting Data on the Clipboard**

The following three predicates are used to put something on the clipboard:

```
cb_PutString(String)
cb_PutPicture(Picture)
cb_PutMetafile(MetaFileHandle)
cb_PutBin(FormatName, Binary)
```

Any previous clipboard content will be lost.

**Testing for Data on the Clipboard**

There is a predicate to test for each type of data on the clipboard. To test if a
binary is available, a name for the format must be supplied.

```
cb_StringAvailable()
cb_PictureAvailable()
cb_MetafileAvailable()
cb_BinAvailable(FormatName)
```

Applications often want to enable or disable the Paste menu item in the Edit
menu depending upon whether or not there is something suitable on the
clipboard. There is a special event *e_InitMenu*, which is sent when the menu is
activated. This means, that it is only necessary to change the menu, when this
event is sent.

```
ehandler(Win, e_InitMenu(), 0):-
    cb_StringAvailable(),
    menu_Enable(Win, menu_editPaste, b_True), !.

ehandler(Win, e_InitMenu(), 0):-
    menu_Enable(Win, menu_editPaste, b_False).
```
CHAPTER 19

Carets

A caret marks the text insertion point of an edit control or a window. It must not be confused with the cursor, which marks the current screen position of the mouse. Often an application will place the caret by checking for text in the area of a mouse click. When the mouse is moved by the user the caret stays put. The caret is often referred to as the text insertion point. Each window has its own caret with its own position.

There are three caret handling predicates:

\textit{caret\_Set(Window, X, Y)}

Moves the caret of \textit{Window} to the coordinate given by \textit{X} and \textit{Y} (relative to the top left corner of the window).

\textit{caret\_Set(Window, X, Y, Width, Height)}

Shows the caret in the specified \textit{Window} (if it was hidden), sets caret dimensions equal to \textit{Width} and \textit{Height}, and moves it to the coordinate given by \textit{X} and \textit{Y}.

\textit{caret\_Size(Window, Width, Height)}

Defines the caret to be used in the window identified by \textit{Window}, the \textit{Width} and \textit{Height} are given in Logical units.

\textit{caret\_Off(Window)}

Makes the caret of \textit{Window} invisible.
CHAPTER 20

VPI Attribute Flags

The attribute flag identifiers name a number of flag values that can be set or read from the VPI. Some attributes are global for the VPI, others are local to a particular window. A list of attribute identifiers and their descriptions will be found in the following table. While all may be retrieved, the values of those marked 'read only' may not be set from the VPI. Of those which can be set, some must be set prior to calling \texttt{vpi\_Init}, and setting them at any other time will have no effect.

\begin{center}
\begin{tabular}{|l|p{8cm}|}
\hline
\textbf{Attribute Name} & \textbf{Description} \\
\hline
\textbf{Environment:} & \\
\hline
\texttt{attr\_have\_color} & Read-only flag that returns the Boolean value that determines whether the screen can show colors \\
\hline
\texttt{attr\_have\_mouse} & Read-only flag that returns the Boolean value which determines whether there is a mouse connected to the computer \\
\hline
\texttt{attr\_num\_timers} & Read-only flag that returns the maximal number of timers \\
\hline
\texttt{attr\_CheckPortability} & In case this flag is set to \texttt{b\_false} (0), the VPI does not check whether a predicate is supported under the chosen platform. (The correspondent error 6008). Default is \texttt{b\_true} (1). \\
\hline
\textbf{Object Size Attributes:} & \\
\hline
\end{tabular}
\end{center}
<table>
<thead>
<tr>
<th>attr_ctl_button_height</th>
<th>Read-only flag that returns the minimal height of push button controls</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_ctl_checkbox_height</td>
<td>Read-only flag that returns the minimal height of check box controls</td>
</tr>
<tr>
<td>attr_ctl_edit_text_height</td>
<td>Read-only flag that returns the minimal height of edit controls</td>
</tr>
<tr>
<td>attr_ctl_horz_sbar_height</td>
<td>Read-only flag that returns the height of arrow bitmaps on horizontal scroll bars</td>
</tr>
<tr>
<td>attr_ctl_vert_sbar_width</td>
<td>Read-only flag that returns the width of arrow bitmaps on vertical scroll bars</td>
</tr>
<tr>
<td>attr_ctl_radiobutton_height</td>
<td>Read-only flag that returns the minimal height of radio button controls</td>
</tr>
<tr>
<td>attr_ctl_static_text_height</td>
<td>Read-only flag that returns the minimal height of static text controls</td>
</tr>
<tr>
<td>attr_icon_width</td>
<td>Read-only flag that returns the width of icons</td>
</tr>
<tr>
<td>attr_icon_height</td>
<td>Read-only flag that returns the height of icons</td>
</tr>
</tbody>
</table>

**Predefined Windows:**

<table>
<thead>
<tr>
<th>attr_screen_window</th>
<th>Read-only flag that returns the VPI handle of the Screen window</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_task_window</td>
<td>Read-only flag that returns the VPI handle of the Task window</td>
</tr>
</tbody>
</table>

**System Metric Attributes:**

<table>
<thead>
<tr>
<th>attr_screen_height</th>
<th>Read-only flag that returns the height of the display in raster lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_screen_width</td>
<td>Read-only flag that returns the width of the display in pixels</td>
</tr>
<tr>
<td>attr_screen_hres</td>
<td>Read-only flag that returns the number of pixels per logical inch along the display width</td>
</tr>
<tr>
<td>attr_screen_vres</td>
<td>Read-only flag that returns the number of pixels per logical inch along the display height</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>attr_printer_height</td>
<td>Read-only flag that returns the height of the being printed page in raster lines</td>
</tr>
<tr>
<td>attr_printer_width</td>
<td>Read-only flag that returns the width of the being printed page in pixels</td>
</tr>
<tr>
<td>attr_printer_hres</td>
<td>Read-only flag that returns the number of pixels of the being printed page per logical inch along the page width</td>
</tr>
<tr>
<td>attr_printer_vres</td>
<td>Read-only flag that returns the number of pixels of the being printed page per logical inch along the page height</td>
</tr>
</tbody>
</table>

**Window Metric Attributes:**

<table>
<thead>
<tr>
<th>attr_docframe_width</th>
<th>Read-only flag that returns the width of the window frame that can be sized</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_docframe_height</td>
<td>Read-only flag that returns the height of the window frame that can be sized</td>
</tr>
<tr>
<td>attr_frame_width</td>
<td>Read-only flag that returns the width of the window frame that can not be sized</td>
</tr>
<tr>
<td>attr_frame_height</td>
<td>Read-only flag that returns the height of the window frame that can not be sized</td>
</tr>
<tr>
<td>attr_dblframe_width</td>
<td>Read-only flag that returns the frame width when window has the <code>wsf_DlgBorder</code> style</td>
</tr>
<tr>
<td>attr_dblframe_height</td>
<td>Read-only flag that returns the frame height when a window has the <code>wsf_DlgBorder</code> style</td>
</tr>
<tr>
<td>attr_menu_height</td>
<td>Read-only flag that returns the height of single line menu bars. This is the menu height minus the height of the window frame that can not be sized</td>
</tr>
<tr>
<td>attr_title_height</td>
<td>Read-only flag that returns the height of a window title. This is the title height plus the height of the window frame that cannot be sized</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>System-defined Window Colors:</strong></td>
<td></td>
</tr>
<tr>
<td>attr_color_activeborder</td>
<td>Read-only flag that returns an active window border color</td>
</tr>
<tr>
<td>attr_color_activecaption</td>
<td>Read-only flag that returns an active window title color</td>
</tr>
<tr>
<td>attr_color_appworkspace</td>
<td>Read-only flag that returns a background color of multiple document interface (MDI) applications</td>
</tr>
<tr>
<td>attr_color_desktop</td>
<td>Read-only flag that returns a desktop color</td>
</tr>
<tr>
<td>attr_color_btnface</td>
<td>Read-only flag that returns a face shading color on push buttons</td>
</tr>
<tr>
<td>attr_color_btnshadow</td>
<td>Read-only flag that returns an edge shading color on push buttons</td>
</tr>
<tr>
<td>attr_color_btntext</td>
<td>Read-only flag that returns a text color on push buttons</td>
</tr>
<tr>
<td>attr_color_captiontext</td>
<td>Read-only flag that returns the text color in title bars, size buttons, scroll-bar arrow buttons</td>
</tr>
<tr>
<td>attr_color_graytext</td>
<td>Read-only flag that returns a grayed (dimmed) text color. This color is zero if the current display driver does not support a solid gray color</td>
</tr>
<tr>
<td>attr_color_highlight</td>
<td>Read-only flag that returns the background color of the selected item in a control</td>
</tr>
<tr>
<td>attr_color_highlighttext</td>
<td>Read-only flag that returns a text color of the selected item in a control</td>
</tr>
<tr>
<td>attr_color_inactiveborder</td>
<td>Read-only flag that returns a color of an inactive window border</td>
</tr>
<tr>
<td>attr_color_inactivecaption</td>
<td>Read-only flag that returns the title color of inactive windows</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>attr_color_inactivecaptiontext</td>
<td>Read-only flag that returns the title text color of inactive windows</td>
</tr>
<tr>
<td>attr_color_menu</td>
<td>Read-only flag that returns the background color of menus</td>
</tr>
<tr>
<td>attr_color_menustring</td>
<td>Read-only flag that returns the text color in menus</td>
</tr>
<tr>
<td>attr_color_scrollbar</td>
<td>Read-only flag that returns the gray area color of scroll-bars</td>
</tr>
<tr>
<td>attr_color_windowtext</td>
<td>Read-only flag that returns the system text color in the client area of windows</td>
</tr>
<tr>
<td>attr_color_window</td>
<td>Read-only flag that returns the system background color in the client area of windows</td>
</tr>
<tr>
<td>attr_color_windowframe</td>
<td>Read-only flag that returns the frame color of windows</td>
</tr>
<tr>
<td>attr_color_windowstatictext</td>
<td>Read-only flag that returns the text color in static text controls</td>
</tr>
<tr>
<td>attr_color_tooltiptext</td>
<td>Read-only flag that returns the text color in tool tips under Windows 95. Under other platforms it returns 0</td>
</tr>
<tr>
<td>attr_color_tooltipback</td>
<td>Read-only flag that returns the background color of tool tips under Windows 95. Under other platforms it returns 0</td>
</tr>
</tbody>
</table>

**Window Attributes:**

<table>
<thead>
<tr>
<th>attr_native_window</th>
<th>Read-only flag that returns the native handle of a window</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_win_instance</td>
<td>Read-only flag that returns the instance handle of the application</td>
</tr>
<tr>
<td>attr_win_mdi</td>
<td>Under Windows attr_win_mdi is used to set the MDI mode.</td>
</tr>
<tr>
<td>attr_win_mdi_client_hwnd</td>
<td>Read-only flag. Under Windows it returns the native handle of the MDI task window. Under OS/2 it always returns 0.</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>attr_win_tbar</td>
<td>Flag that is used to set the height of the tool bar area on the top of a window</td>
</tr>
<tr>
<td>attr_win_lbar</td>
<td>Flag that is used to set/get the width of the tool bar area on the left side of a window</td>
</tr>
<tr>
<td>attr_win_rbar</td>
<td>Flag that is used to set/get the width of the tool bar area on the right side of a window</td>
</tr>
<tr>
<td>attr_win_sbar</td>
<td>Flag that is used to set the height of the status line area in the bottom of a window</td>
</tr>
<tr>
<td>attr_win_3dcontrols</td>
<td>Flag that is used to turn on/off support for 3d-controls. <strong>Note:</strong> this flag must be set prior to calling <code>vpi_Init</code>.</td>
</tr>
<tr>
<td>attr_suppress_update_check</td>
<td>Flag that is used to suppress the <code>modified()</code> notification for edit controls.</td>
</tr>
<tr>
<td>attr_erase_background</td>
<td>If this flag is set to <code>b_False</code>, there will be no automatic erasing of the background for windows.</td>
</tr>
</tbody>
</table>

The current value of any of these attributes may be retrieved at any time by:

```c
LONG = vpi_GetAttrVal(LONG Attribute)
```

or for attributes connected to the windows

```c
LONG = win_GetAttrVal(WINDOW, LONG Attribute)
```

Since the value is returned as a long, it is sometimes necessary to use the CAST operation to turn the value into the proper domain. E.g. to retrieve the Task window handle, the following clause can be used:

```c
TaskWin = cast(WINDOW, vpi_GetAttrVal(attr_task_window)),
```

The attributes, which are not read-only, can be changed by a call to:

```c
vpi_SetAttrVal(LONG Attr, LONG Value)
```

There are currently no attributes specific to a given window that can be changed.
CHAPTER 21

Window Classes

Creating window classes is supported in the VPI. A window class can be used to implement a custom control, which can be put into a dialog.

Creating a class can be done by a call to:

```c
class_Create(STRING ClassName, EHANDLER)
```

And the class can be removed by a call to:

```c
class_Destroy(STRING ClassName)
```

Make sure that a class is created before using a dialog with custom controls. When a class is created, there is a normal Prolog event-handler predicate associated with the class. When a dialog is created, the window for the custom control is automatically created, and the event-handler predicate will receive an `e_Create` event for the class.

When `class_Create` is called in a dynamic-link library (DLL), it creates a global window class. The operating system loads the specified DLLs in the context of the newly started process before calling the `Goal` in that process, a DLL registers global window classes during its initialization procedure.

As an example, the following piece of code can be used to implement a control that displays a picture in a dialog:

GLOBAL PREDICATES
```
  bmpctrl_register
  bmpctrl_SetBMPFile(WINDOW, String File, BOOLEAN Stretch) - (i,i,i)
```

DOMAINS
```
CTRLBMP = file(STRING FileName); resid(RESID)
```

DATABASE - bmpctrl
```
  ctrl_bmp(WINDOW, CTRLBMP, BOOLEAN Stretch)
```

PREDICATES
```
  win_bmpctrl_eh : EHANDLER
```
Note also, that the predefined tools like the Tree tool, and the editor can also be put into a custom control. For this purpose there are some predicates: `edit_CreateSubClass` and `tree_CreateSubClass` that must be called in the `e_Create` event for the subclass. Please study the `VPI\EXAMPLES\CLASS` for an example on how to create tree and editor subclasses.

If you create any tools, which can be put into a custom control, you should use the same mechanism as is used in the Tree tool. The trick is that the Tree tool changes the event-handler for the window, remembers the event-handler for the subclass, and passes the events back to the subclass:

Note also, that the predefined tools like the Tree tool, and the editor can also be put into a custom control. For this purpose there are some predicates: `edit_CreateSubClass` and `tree_CreateSubClass` that must be called in the `e_Create` event for the subclass. Please study the `VPI\EXAMPLES\CLASS` for an example on how to create tree and editor subclasses.

If you create any tools, which can be put into a custom control, you should use the same mechanism as is used in the Tree tool. The trick is that the Tree tool changes the event-handler for the window, remembers the event-handler for the subclass, and passes the events back to the subclass:

Note also, that the predefined tools like the Tree tool, and the editor can also be put into a custom control. For this purpose there are some predicates: `edit_CreateSubClass` and `tree_CreateSubClass` that must be called in the `e_Create` event for the subclass. Please study the `VPI\EXAMPLES\CLASS` for an example on how to create tree and editor subclasses.

If you create any tools, which can be put into a custom control, you should use the same mechanism as is used in the Tree tool. The trick is that the Tree tool changes the event-handler for the window, remembers the event-handler for the subclass, and passes the events back to the subclass:
tree_after_handling(Win, e_Destroy()) :-
  tree_ehandler(Win, e_Destroy()).

tree_after_handling(Win, e_Size()) :-
  tree_ehandler(Win, e_Size()).

tree_after_handling(_, _).

Controls Subclassing

Normally, all controls from the same control class share the same window event-handler. A subclass is a control or set of controls that belong to the same predefined control class (standard types of controls), and whose events are intercepted and processed by another window event-handler before being passed to the default event-handler of the predefined control class.

To create a subclass, call the `win_SetSubClassHandler` predicate to associate the new subclass event-handler with a particular control. For this control VPI first will pass each event to the specified subclass event-handler and then, if an event is not caught by the subclass event-handler or the event-handler fails for the event, will pass the event to the default event-handler of the primary predefined control class. In VPI it is possible to create subclass only to the predefined control classes (not to custom controls and not to ordinary windows or dialogs).
Implementing On-line Help

The VPI supports the following predicates to activate the on-line help. See the Guided Tour example for how to get support from the VDE to do on-line help.

\[
\begin{align*}
\text{vpi\_ShowHelp} & (\text{STRING FileName}) \\
\text{vpi\_ShowHelpContext} & (\text{STRING FileName, LONG ContextId}) \\
\text{vpi\_ShowHelpKeyWord} & (\text{STRING FileName, STRING KeyWord}) \\
\text{vpi\_HelpClose} & ()
\end{align*}
\]
CHAPTER 23

Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) is used to communicate with other applications. The communication can either be a single data transfer or it can establish a communication link between two tasks. For example, it is possible to send a command to a word processor for it to load a text file, to read the contents of a spreadsheet cell, or communicate with a database server.

The DDE support provided in Visual Prolog is based on the DDEML (Dynamic data Exchange Management Library) library on the MS-Windows platform. It is currently not portable to the other platforms.

DDE communication supported by Visual Prolog is fully compatible with other MS Windows applications. While DDE communication can be done between two Prolog applications, you can also use it to communicate with applications like MS Excel, MS Word, the Program Manager etc. There is a further advantage when using Windows NT/2000, Windows 95/98/ME or Windows 3.11 (Windows for work groups). The data exchange can run over a network through NETDDE to implement sophisticated network-based client server computing.

Usually an application is either a client or a server. Some applications (e.g. MS Word) can take both client and server roles.

The Client Role

Communication is always initiated and controlled by the client.

A client program displays the information obtained from a server in an application dependent way. To obtain information it contacts a server and sends commands or makes a request for data.

The client can set up an advisory loop, where the server can send data asynchronously to the client.

A client can request data from multiple servers simultaneously.

Clients often need to send messages to other clients; this is usually done by using a (common) server as a message router.
The Server Role

A server program responds to requests from clients by obtaining information and formatting this information to make it usable by the client.

A server will register the service name(s) that it supports.

A server can have many clients simultaneously.

Client and Server Interaction

The DDE client initiates the communication by establishing a conversation with a DDE server, which responds by sending a conversation handle to the client. Both parties can terminate the conversation at any time.

All DDE messages are sent as \textit{e_DDE} events that the application receives in its Task window event-handler.

For example, when a client application attempts to establish a conversation with a server application, the client calls the \textit{dde_Connect} function. This will result in the arrival of \textit{e_DDE}(\textit{connect}) event in the server's Task window event-handler. The server can allow the connection by returning \textit{b_True}, or it can deny the connection by returning \textit{b_False} from the event-handler.

Service/Topic/Items

When a client application begins a DDE conversation, it must specify two things:

A Service name, which usually is the name of the application to talk to: Microsoft Access, Microsoft Excel, Microsoft Word for Windows, Program Manager, etc.

A Topic name, which is the subject of the conversation. Many programs like Word for Windows and Excel uses the name of the file as the topic "REPORT.DOC" or "BUDGET.XLS". A special topic that many applications support is the "system" topic. The System topic can usually be used to query which other topics are available.

Once the conversation is established, it is not possible to change the Service and the Topic. But the conversation might refer to several different pieces of information by referring to different items. In Excel a Cell is the Item (Such as R1C2), in Word a Bookmark can be the Item, in Access, a record in the database is the Item.
Initializing and Closing Down the DDE

An application must initialize the DDE before use. This is best done in the e_Create event of the applications Task window:

```prolog
task_win_eh(_Win, e_Create(_), 0):-
    dde_Init([dde_clientonly]),
    ... .
```

The dde_Init predicate takes a list of flags. Clients should always just specify ddef_clientonly. For servers, there are many other possibilities.

When the application terminates it must call the dde_End predicate to close down the DDE system:

```prolog
task_win_eh(_Win, e_Destroy, 0):-
    dde_End().
```

Simple DDE Client Operations

Most applications will just use simple DDE clients to communicate with other standard applications like the Program Manager, Word for Windows and Excel, so most people can just read the following information about the DDE Client operations, which is quite simple to use. If you need to implement full client/server architecture you need to read the whole DDE chapter.

Connecting to a Server Application

After initializing its DDE the client can attempt to connect to a server by using the dde_Connect predicate:

```prolog
DDE_CONV dde_Connect(STRING Service, STRING Topic)
```

It takes a Service and a Topic name, and if the connection is successful, it returns a conversation handle, if it is not successful it exits with a runtime error.

The communication channel can be closed down with a call to dde_DisConnect:

```prolog
dde_DisConnect(DDE_CONV Conv)
```

A little example on how to connect to Word for Windows and disconnect again are:
task_win_eh(_Win, e_Menu(id_Connect_Word, _), 0):-!,
  trap(Conv = dde_Connect("WinWord", "System"), _,
       write("WinWord not running\n")),
  dde_DisConnect(Conv).

Executing a Command on the Server Application

Some servers may allow clients to issue execute command strings. The dde_Execute predicate is provided for this purpose.

DDE_TRANSID dde_Execute(DDE_CONV Conv, STRING Command, LONG Timeout)

The timeout value, and the transaction identifier will be explained later.

As example the following clause is used to connect to Word for Windows and send it a command to load the file DDETEST.DOC from the current directory of the application:

task_win_eh(_Win, e_Menu(id_word_open_ddetest, _ShiftCtlAlt), 0):-!,
  trap(Conv = dde_Connect("WinWord", "System"), _,
       write("WinWord not running\n")),
  filenamepath(FullName, "", "ddetest.doc"),
  format(Command, "[FileOpen .Name="%"]", FullName),
  write("Command = ", Command), nl,
  dde_Execute(Conv, Command, 0),
  dde_DisConnect(Conv),

The available commands and their syntax vary from application to application.

Requesting Data from a Server Application

The predicate dde_request can be used to retrieve data from the server:

DDE_TRANSID dde_Request(DDE_CONV Conv, STRING Item,
                        LONG TimeOut, DDE_FORMAT, DDE_DATA Val)

Basically it takes a handle to an open conversation and the name of an Item, and returns the requested data. The predicate also takes a timeout value, that indicates how long to wait before timing out, and there is the possibility to specify what kind of data to request, for most DDE operations this should just be set to the value ddecf_text which gives a regular text string. The purpose of the returned transaction identifier will be explained under the more advances transaction handling.
The following clause will both establish a connection with Excel and retrieve the value at Row1, Column1:

```
(task_win_eh(_Win, e_Menu(id_excel_peek, _ShiftCtlAlt),0):-!,
  trap(Conv = dde_Connect("Excel","[ddetest.xls]Sheet1"),_,,
       write("ddetest.xls not loaded\n")),
  dde_Request(Conv, "R1C1", 1000, ddecf_Text, Val),
  write("Got: ", Val), nl,
  dde_DisConnect(Conv).
```

### Sending Data to a Server Application

The predicate `dde_Poke` is used to send data to another application:

```
DDE_TRANSID dde_Poke(DDE_CONV Conv, STRING Item,
                    DDE_DATA Val, DDE_FORMAT, LONG TimeOut)
```

As example, the following clause will open a DDE connection and put a value in a spreadsheet cell:

```
(task_win_eh(_Win, e_Menu(id_excel_poke, _ShiftCtlAlt),0):- !,
  trap(Conv = dde_Connect("Excel","[ddetest.xls]Sheet1"),_,,
       write("ddetest.xls not loaded\n")),
  dde_Poke(Conv, "R1C1", str(Val),ddecf_text, 0),
  dde_DisConnect(Conv),
```

### More Advanced Client Operations

When the goal is to establish a true client/server architecture things become a little bit more complicated.

### Transactions

The previous example on requesting data from the server by `dde_Request` was a synchronous request; the client application was waiting until the data arrived from the server. It is often not desirable to wait for operations to complete, the system must be fully responsive, and cannot accept to block the whole application until the server responds.

If blocking is not accepted, asynchronous operation must be used instead. In this case a timeout value of zero is used and the `dde_Request` predicate will return immediately after the call with a transaction identifier instead of data. The data will then arrive sometime later as an event identified by the returned transaction identifier.
identifier. The disadvantage of using asynchronous programming is that it is a bit more complicated.

**Transaction Events**

When the data arrives to the client, it will arrive as an \texttt{e\_DDE} event to the Task window event-handler:

\begin{verbatim}
task_win_eh(_Win, e_DDE(_Conv, 
   trans_complete(_Topic, _Item, _TRANSID, _DDE_DATA)),0):-!, 
   retract(trid(_Conv, _TRANSID)),!, 
   DDE_DATA = str(Data), 
   write("Received from a server (asynchronous): ",Data),nl.
\end{verbatim}

**Abandoning an Asynchronous Data Request**

A Client can call the \texttt{dde\_AbandonTrans()} predicate if it no longer needs the data resulting from an asynchronous request.

\begin{verbatim}
dde_AbandonTrans(DDE_CONV Conv, DDE_TRANSID Id)
\end{verbatim}

This call should only be issued by a client. After issue any arriving asynchronous data (for the conversation that issued the \texttt{dde\_AbandonTrans()}) will be ignored.

**Transferring Data between the Server and a Client**

The data that are sent between the server and the client belong to the DDE\_DATA domain. The DDE\_DATA domain is defined as follows:

\begin{verbatim}
DDE\_DATA = str\(\text{STRING}\); bin\(\text{BINARY}\); pict\(\text{PICTURE}\); nodata
\end{verbatim}

The form \texttt{nodata} is returned in all the situations when a timeout value of zero is requested, in this case, the real data arrives later in the \texttt{trans_complete} event.

Since the program in the other end might not be a Visual Prolog application, the DDE supports all formats sent under DDE. This is done by giving the requested format in the \texttt{dde\_Request} call. The possible formats under Windows are:

\begin{verbatim}
ddefcf\_Text, ddefcf\_Bitmap, ddefcf\_Metafilepict, ddefcf\_Sylk, ddefcf\_DIF, 
ddefcf\_Tiff, ddefcf\_OMText, ddefcf\_Dib, ddefcf\_Palette, ddefcf\_PenData, 
ddefcf\_Riff, ddefcf\_Wave
\end{verbatim}

When using \texttt{ddefcf\_text} a \texttt{str\(\text{STRING}\)} is returned, and when using \texttt{ddefcf\_Bitmap} a \texttt{pict\(\text{PICTURE}\)} is returned, for all other formats \texttt{bin\(\text{BINARY}\)} is returned.
Restarting a Conversation that has been Terminated

The `dde_ReConnect()` predicate can be used to attempt to reestablish a conversation with a service that has terminated a conversation with the client.

```
dde_ReConnect(DDE_CONV Conv)
```

When the conversation is reestablished, the DDEML attempts to reestablish any preexisting advise loops.

Multiple Connections

A client can obtain connections to all servers that support a given service and topic by using the `dde_ConnectMult()`. This predicate returns a list of conversation handles.

```
DDE_CONVLIST dde_ConnectMult(Service, Topic, DDE_CONVMULT)
```

The argument DDE_CONVMULT returns a special handle that must be used to close down the connection after use.

```
dde_DisConnectMult(DDE_CONVMULT ConvMult)
```

Wild Connections

The predicate `dde_Connect` has a special feature to connect to all topics for a given service. In this case, the topic parameter should be an empty string.

```
_Dde_Conv = dde_Connect("Myserv", "")
```

The DDE Server Operations

Initializing a Server Application

When calling `dde_Init` in a server application, there are additional relevant flags. These all serve to skip unnecessary events in order to optimize the application’s performance.
Table Error! Bookmark not defined..20: Initializing a Server Application

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddef_fail_selfconnections</code></td>
<td>Prevents an application from connecting to itself</td>
</tr>
<tr>
<td><code>ddef_fail_connections</code></td>
<td>Prevents the application from receiving any connects</td>
</tr>
<tr>
<td><code>ddef_fail_advises</code></td>
<td>Prevents the application from receiving any advisory loop start and stop commands</td>
</tr>
<tr>
<td><code>ddef_fail_executes</code></td>
<td>Prevents the application from receiving any execute commands</td>
</tr>
<tr>
<td><code>ddef_fail_pokes</code></td>
<td>Prevents the application from receiving any poke commands.</td>
</tr>
<tr>
<td><code>ddef_fail_requests</code></td>
<td>Prevents the application from receiving any request commands</td>
</tr>
<tr>
<td><code>ddef_fail_allsvrxactions</code></td>
<td>Prevents the application from receiving any server events. (Combine all the <code>ddef_fail*</code> events)</td>
</tr>
<tr>
<td><code>ddef_skip_connect_confirms</code></td>
<td>Prevents the application from receiving any <code>connect_confirmation</code> events</td>
</tr>
<tr>
<td><code>ddef_skip_disconnects</code></td>
<td>Prevents the application from receiving any disconnect events</td>
</tr>
<tr>
<td><code>ddef_skip_allnotifications</code></td>
<td>Prevents the application from receiving any notification events. (Combine all the <code>ddef_skip</code> flags)</td>
</tr>
</tbody>
</table>

In addition the Server application must register each service that it supports by calling the `dde_RegisterService` predicate:

```
dde_RegisterService(STRING Service, BOOLEAN Register)
```

As an example, look at:

```
task_win_eh(_Win, e_Create(_),0):-!,
   dde_Init([ddef_fail_pokes]),
   dde_RegisterService("Myserv", b_true).
```
The service must be unregistered at the termination of the application:

```prolog
task_win_eh(_Win, e_Destroy, 0):-!,
dde_RegisterService("Myserv", b_false),
dde_End().
```

**Accepting a Connection from a Client**

When a client requests a connection, the sever receives a connect message in its DDE event-handler. The server accepts the connection by returning a non-zero value in the last parameter, a value of zero (or \texttt{b\_False}) rejects the request. The first parameter of the DDE submessage (usually) contains a unique conversation identifier (handle) that both parties must use when conducting the conversation. However, in the servers connect event this variable (_\texttt{Conv}) will not contain a valid conversation handle and must not be used.

```prolog
task_win_eh(_Win, e_DDE(_Conv, connect("service", "topic")), b_True):-!,
write("connect() message is received and accepted"),nl,
```

As noted above the conversation handle that a server receives in a DDE connect submessage must not be used in conversations. The "real" handle arrives after the connection is established. A \texttt{connect\_confirm} event is generated by the system. This event can be used to save the conversation handle in a database:

```prolog
task_win_eh(_Win, e_DDE(_Conv, connect\_confirm("service", "topic")), 0):-
assert(valid_dde\_handle(_Win, _Conv, "service", "topic")),
write("connect\_confirm() message is received"),nl,
```

When the conversation is finally terminated by the client, the server will receive a \texttt{disconnect} event:

```prolog
task_win_eh(_Win, e_DDE(_Conv, disconnect()), 0):-!,
retract(conv(_Conv)),
```

**Accepting a Command from a Client**

When a client sends a command string by calling dde_Execute, the server will receive an \texttt{execute()} message:

```prolog
task_win_eh(_Win, e_DDE(_Conv, execute(_Topic, _Command)), Ret):-!,
```

The server must take appropriate actions for the command, and return of the following values:
Table Error! Bookmark not defined..21: Constants for Accepting a Command from a Client

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dderes_fnotprocessed</td>
<td>This value should be returned if the command could not be processed</td>
</tr>
<tr>
<td>dderes_fbusy</td>
<td>This value should be returned if the server is too busy to handle the command</td>
</tr>
<tr>
<td>dderes_fack</td>
<td>This value should be returned if the command are properly executed</td>
</tr>
</tbody>
</table>

Accepting Data from a Client

When the client sends data to the server by calling the dde_Poke predicate, the server will in the other end receive a poke message:

```prolog
task_win_eh(_Win, e_DDE(_Conv, 
    poke("Mytopic", "Myitem", _DDE_DATA),dderes_fack):-!, 
    _DDE_DATA = str(S), 
    write("Received from a client: ",S),nl,
```

The return value must be dderes_fack, dderes_fbusy or dderes_fnotprocessed.

Responding to a Data Request

When a client issues a dde_Request command, the server will receive a request event, and must return data in response to this event. This gives a little problem in Prolog because the return value is a long value, and the DDE_DATA is allocated on the Prolog Global Stack. As a work-around for this, a special predicate has to be used to prepare the return data:

```prolog
LONG dde_MakeDDEReturnData(DDE_DATA Val)
```

A typical clause to return data to the client will then be:

```prolog
task_win_eh(_Win, e_DDE(_Conv, request("Mytopic", "Myitem")),DATA):-!, 
    DATA = dde_MakeDDEReturnData(str("Hello from a Server")),
```
Advisory Loops

A client application can use the DDE to establish one or more permanent links to items in a server application. When such a link is established, the server sends periodic updates about the linked item to the client (typically, whenever the value of the item associated with the server application changes). This establishes an advise loop between the two applications that remains in place until the client ends it. The advisory loop is started when the client calls the `dde_StartAdv` predicate:

```c
DDE_TRANSID dde_StartAdv(DDE_CONV Conv, STRING Item, DDE_FORMAT,
                          DDE_ADVFLAGS, LONG TimeOut)
```

The DDE_ADVFLAGS is a list that can contain the following two values:

- **ddeadv_NoData**: Instructs the server to notify the client of any data changes without actually sending the data. This flag gives the client the option of ignoring the notification or requesting the changed data from the server. When the flag is specified the connection is usually referred to as a Warm link instead of a Hot link when the data are actually sent.

- **ddeadv_AckReq**: Instructs the server to wait until the client acknowledges that it received the previous data item before sending the next data item. This flag prevents a fast server from sending data faster than the client can process it.

The advise loop can be stopped by the client calling the `dde_StopAdv` predicate:

```c
DDE_TRANSID dde_StopAdv(DDE_CONV Conv, STRING Item,
                         LONG TimeOut)
```

After the client has successfully called the `dde_StartAdv` predicate, the server will receive an **advstart** event. In response to this event, the server must set up some mechanism so it can inform the client when the requested item has changed.

In the following example clause, a timer is set up, so each time this triggers a new value is sent back to the client.

```prolog
task_win_eh(_Win, e_DDE(_Conv, advstart("MYtopic", "Myitem")),1):-!,
clear_timers,
TIMERID = timer_Set(_Win, 2000),
assert(tim(TIMERID)),
write("Start advise loop"),
```

Visual Programming Interface
Similar, the server will receive an `advstop` event when the advisory loop is terminated by the client:

```prolog
task_win_eh(_Win, e_DDE(_Conv, advstop("MYtopic", "Myitem")),1):-!,
    clear_timers,
```

When the server wants to send data for the advise loop back to the client, it is done indirectly by calling the `dde_PostAdvise` predicate.

```prolog
dde_PostAdvise(STRING Topic, STRING Item)
```

As example from the previous example where a timer was set up:

```prolog
task_win_eh(_Win, e_Timer(_TimerId),0):-!,
    dde_PostAdvise("MYtopic", "Myitem").
```

The result of calling the `dde_PostAdvise` predicate will be that the server receives an `advreq` event, and the data must be returned to the client in response to this event:

```prolog
task_win_eh(_Win, e_DDE(_Conv, advreq("MYtopic", "Myitem")),DATA):-!,
    time(HH, MM, SS, _CC),
    format(Msg, "Time is: (%02:%02:%02)", HH, MM, SS),
    MyData = str(Msg),
    DATA = dde_MakeDDEReturnData(MyData).
```

On the client side, the client receives the data in an `advdata` event:

```prolog
task_win_eh(_WIN, e_DDE(_Conv, advdata(_Topic, _Item, DATA)),0):-!,
    DATA = str(Str),
    write("Received: ",Str),nl,
```

### Accepting a Wild Connection

DDE clients can make a wild connection, which means, that it will connect to all topics for a given service. In response to this, the server must return a structure with a combination of Service/Topic pairs.

```prolog
task_win_eh(_, e_DDE(_Conv, wildconnect("Myserv",_Topic)),ServiceInfo):-
    ServiceInfo = dde_MakeDDEReturnServiceInfo( 
    [service_info("Myserv", "Mytopic"),service_info("Myserv", "System")]).
```

After the server has returned the list of Service/Topic pairs, it will receive a `connect_confirm` for each pair.
Temporarily Queuing DDE Events

The `dde_EnableEvents` predicate can be used to queue a conversation event temporarily.

\[
dde_EnableEvents(DDE_CONV Conv, BOOLEAN EnableDisable)
\]

Giving a value of `b_FALSE` will queue up all DDE events, giving a value of `b_TRUE` will enable the DDE events again. Normal transmission of events should be enabled as soon as possible.

Critical Errors

An `error()` event is issued to a client or to a server if a critical error occurs during the DDE conversations.

Table of DDE Predicates/DDE Events

The following table lists the available predicates and (where applicable) the events received by the other party:

<table>
<thead>
<tr>
<th>DDE PREDICATES</th>
<th>DDE EVENTS (Received by the other part)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client:</strong></td>
<td></td>
</tr>
<tr>
<td><code>dde_AbandonTrans</code></td>
<td></td>
</tr>
<tr>
<td><code>dde_Connect</code></td>
<td><code>connect(STRING Service, STRING Topic)</code></td>
</tr>
<tr>
<td></td>
<td><code>connect_confirm(STRING Service, STRING Topic)</code></td>
</tr>
<tr>
<td><code>dde_ConnectMult</code></td>
<td><code>wildconnect(STRING Service, STRING Topic)</code></td>
</tr>
<tr>
<td><code>dde_DisConnectMult</code></td>
<td></td>
</tr>
<tr>
<td><code>dde_EnableEvents</code></td>
<td></td>
</tr>
<tr>
<td><code>dde_Execute</code></td>
<td><code>execute(STRING Topic, STRING Command)</code></td>
</tr>
<tr>
<td><code>dde_Poke</code></td>
<td><code>poke(STRING Topic, STRING Item, DDE_DATA Data)</code></td>
</tr>
<tr>
<td><code>dde_ReConnect</code></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>dde_Request</td>
<td>request(STRING Topic, STRING Item)</td>
</tr>
<tr>
<td>dde_StartAdv</td>
<td>advstart(STRING Topic, STRING Item)</td>
</tr>
<tr>
<td>dde_StopAdv</td>
<td>advstop(STRING Topic, STRING Item)</td>
</tr>
</tbody>
</table>

**Server:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dde_MakeDDEReturnData</td>
<td></td>
</tr>
<tr>
<td>dde_MakeDDEReturnServiceInfo</td>
<td></td>
</tr>
<tr>
<td>dde_RegisterService</td>
<td></td>
</tr>
<tr>
<td>dde_PostAdvise</td>
<td>advreq(STRING Topic, STRING Item)</td>
</tr>
<tr>
<td></td>
<td>advdata(STRING Topic, STRING Item, DDE_DATA Data)</td>
</tr>
<tr>
<td></td>
<td>trans_complete(STRING Topic, STRING Item, DDE_TRANSID Id, DDE_DATA Data)</td>
</tr>
</tbody>
</table>

**Client/Server:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dde_Init</td>
<td></td>
</tr>
<tr>
<td>dde_DisConnect</td>
<td>disconnect();</td>
</tr>
<tr>
<td>dde_End</td>
<td>error()</td>
</tr>
</tbody>
</table>

Many of the predicates used generate blocking calls. They can however be used to obtain data in a non-blocking manner, namely the advisory loop (warm and hot links).

**NETDDE**

Microsoft's NETDDE.DLL extends DDE capability across to applications running on a Microsoft LAN.

To use NETDDE the server program must be registered as being sharable. The client program uses a special form of `dde_Connect`, which includes the network name of the machine running the server application.
The format for the NETDDE version of the `dde_Connect` command is:

```
DDE_CONV = dde_Connect("\\server_machines_network_name\NDDE$", "dummy_topic")
```

The topic that is accessed will be the one defined in the DDESHARE registration.

The Microsoft Windows for Workgroups Software Development Kit (SDK) contains a program called DDESHARE.EXE, which can be used to easily set up DDE shares.

A server registered via DDE share can be automatically started; if it is not running when a client requests a connection.

To run NETDDE programs in Windows95, you must add a shortcut for NETDDE.EXE to the Startup group. To do so, use the following four steps:

1. Use the right mouse button to click on an empty space on the task bar, and then click Properties on the menu that appears.
2. On the Start Menu Programs tab, click Add.
3. Use the Create Shortcut Wizard to create a shortcut for NETDDE.EXE in the Windows folder.
4. After you create the shortcut, restart your computer.
Metafile Handling

As an alternative to storing pictures in bitmaps, MS-Windows can store pictures in metafiles. A metafile is a recording of the actual drawing operations necessary to create a picture. In the file there are structures that contain enough information for each drawing operation. The picture can be created at a later time by playing these drawing operations in a window.

An advantage with metafiles is that they are completely device independent. When a bitmap is scaled to a printer, each dot needs to be scaled. But for a metafile, the drawings are done in the new device context. A disadvantage is that metafiles are not portable across platforms. Depending on the actual pictures, there might also be substantial speed difference between drawing them as metafiles or as bitmaps.

On Win16, Visual Prolog supports the "Aldus Placeable Metafile" format, which means that the file stores some extra information about the size of the original picture.

Load a Metafile from a File

A metafile must be loaded into memory before it can be drawn in a window, and the predicate \texttt{mf\_Load} does this. Remember to call \texttt{mf\_Destroy} after you are done with the metafile to release the associated memory.

\begin{align*}
\text{METAFILE } & \texttt{mf\_Load(STRING FileName)} \\
\text{METAFILE } & \texttt{mf\_Load(STRING FileName, RCT RctMf, UNSIGNED Resolution)}
\end{align*}

Play a Metafile in a Window

When a metafile is loaded into memory, it can be played in a window or to a printer by a call to \texttt{mf\_Play/3} or \texttt{mf\_Play/2} predicate.

Notice that a metafile keeps the picture frame coordinates in the metafile header. The \texttt{mf\_Play/3} predicate uses this picture frame to map a picture onto the rectangle pointed to by the \textit{Rect} parameter:
mf_Play(WINDOW Win, METAFILE Mf, RCT Rect)

In opposite, mf_Play/2 simply plays a metafile in the rectangle with the picture frame coordinates.

mf_Play(WINDOW Win, METAFILE Mf)

Therefore, to play a metafile in the desired position by mf_Play/2, the programmer should use an appropriate scaling system correlated with the picture frame coordinates. Calling the win_SetMapMode and the win_SetMapScale, before playing the metafile can set up a scaling system.

**Destroy Metafile**

When your application is done with using a metafile, the memory space must be released with a call to:

mf_Destroy(METAFILE Mf)

**Metafile Recording**

It is possible to record a number of drawing operations in a metafile. The steps involved are:

1. Make a call to mf_Open which initiates the recording of drawing operations:

2. WINDOW mf_Open()

3. mf_Open returns a window handle, which must be used for all drawing operations to the metafile

4. Do a number of drawing operations to the metafile window to create a picture.

5. Call mf_Close to complete the recording of the drawing operations and return a handle to the metafile.

6. METAFILE = mf_Close(MetafileWindow)

7. A newly created metafile can be saved in a disk file by a call to mf_Save:

   mf_Save(Metafile, Rct, FileName)

The Rct value is used to indicate the desired size of the drawing in the metafile, so the metafile can be scaled during a later replay.
CHAPTER 25

Error Handling

The VPI provides a safety-net for VPI applications. The event-handling mechanism catches failing event-handlers, exits and run-time errors in the event-handler predicates (or predicates called by them). Therefore, the programmer does not need to create his own event-wrapping predicates as in the earlier Windows version of Prolog.

Argument Validation

In order to make debugging of malfunctioning applications faster, the VPI performs a number of validations on the arguments to VPI predicates. If there are any problems, a run-time error is issued, which reports the source position of the bad call if that information is available. This argument validation takes time, so for well-tested applications it is possible to disable this validation by:

```
  vpi_ArgValidation(BOOLEAN On/Off)
```

Installing a New Error Handler

When an error occurs - and the error is not trapped - the VPI catches the exit in the event-handler and does default error handling with a call to the common dialog `dlg_Error`.

It is possible to install a user-defined error-handling predicate, which might, for example, log the error in a file and request that the user send the log file back to the developer.

The error handler predicate must be declared as belonging to the domain ERRHANDLER and takes two arguments: the window handle and the error number.

A typical error handler in an application might be:

```
PREDICATES
  my_err_handler : ERRHANDLER
```
CLAUSES

    my_err_handler(Win, Err):-
        lasterror(ErrorNo, FileName, IncludeFileName, Position),
        format(S, "ErrorNo=%, FileName=<%>
                 IncludeFileName=<%>, Position=%",
                ErrorNo, FileName, IncludeFileName, Position),
        file_str("error.log", S),
        dlg_Note(
                "Internal error
Please send the file ERROR.LOG to <developer>").

The new error handler can be installed with the call:

    vpi_SetErrorHandler(my_err_handler),

Note that in order to get accurate error positions from the lasterror predicate, the application must be compiled with **Options | Project | Compiler | Errorlevel** set to medium or maximum. The resulting larger application will run a little slower, so in well-tested applications it might be an acceptable trade off to report only the error number so that it can be compiled without diagnostic information being placed in the executable file.
Native API Functions

The VPI handles the most common features found in the native windowing systems. However, for special applications, it might be necessary to go underneath the VPI layer. The VPI layer is essentially a number of functions, which make calls to the underlying native API - it is still possible to call all the underlying API functions directly.

If you need to take advantage of special capabilities of the native operating system you can call the functions of the underlying API (Microsoft Windows, PM etc.) directly. This will make your application non-portable, so it is a good idea to isolate such calls in a module that contains your operating system-dependent code. If you later decide to port the application to another operating system you will only need to rewrite the operating system dependent parts. You can also make use of the automatically defined constants (ws_win, ws_pm etc.) inside ifdef ... enddef constructs to conditionally enable sections of source code based on the windowing system target selected.

Calling Native GUI Functions

In the VPI the native API functions can be declared as global predicates. The language calling conventions to the underlying API functions are different from platform to platform.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Calling Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win16</td>
<td>PASCAL</td>
</tr>
<tr>
<td>Win32</td>
<td>STDCALL</td>
</tr>
<tr>
<td>PM</td>
<td>SYSCALL</td>
</tr>
</tbody>
</table>

For the 16_bit MS-Windows platform, the WINBIND sub-directory contains declarations for most of the native calls. By including WINDOWS.CON, WINDOWS.DOM and WINDOWS.PRE from the WINBIND\INCLUDE directory you can easily call the 16-bit MS-Windows API directly.
Creating a Window outside the VPI

It is possible to create a window by making calls directly to the underlying API, and install event-handler predicates that operate directly on native events.

Native Events for a VPI Window

For a window created with the VPI, the VPI layer only sends relatively few high-level events to the windows event-handler predicate. However, it is possible to receive all events the underlying GUI sends to the window by calling the predicate:

\[
\text{win\_EnableHook}(\text{Win}, \text{BOOLEAN OnOff}),
\]

When this is called with the \texttt{b\_True} parameter for a window the window will start to receive \texttt{e\_Native} events on the form:

\[
\text{my\_event\_handler}(\text{Win}, \text{e\_Native}(\text{Message, WParam, LParam}), 0):-
\]

Where \texttt{Message} in the underlying event identification, \texttt{WParam} is the value (a WORD for 16-bits platforms, a LONG for 32-bits platforms) and \texttt{LParam} is a LONG value. The example in VPI\EXAMPLES\HOOK is an illustration of how this works.

Retrieving the Native Window Handle

A VPI \texttt{Window} argument does actually just pass around the native window handle. This can just be typecast to the correct domain for the native window handle. However, a cleaner and safer method may be instead to use:

\[
\text{NativeWin} = \text{win\_GetAttrVal(Win, attr\_native\_window)},
\]

Retrieving the Native Device Context

It is possible to retrieve the native Device Context (DC) associated with a window by making the call:

\[
\text{DC} = \text{win\_GetNativeGraphicContext(Window,ReleaseFlag)}-(i,o)
\]

The predicate returns a DC, which can be used for drawing operations directly to the native windows API. The flag returns a flag that must be passed back to the call for releasing a device context again:

\[
\text{win\_ReleaseNativeGraphicContext(Window,DC,ReleaseFlag)}
\]
CHAPTER 27

Running External Applications

From your VPI programs you can execute any external program with the call:

\[
\text{ApplicationId} = \text{vpi\_CreateProcess}(\text{ReceiveMsgWin, Program, Param, TaskWinInitFlag})
\]

Here \text{Program} is a string specifying the path and filename of the application to be run. \text{Param} is a string specifying the command line for the application. \text{TaskWinInitFlag} can be one of the following flags: wsf\_Invisible, wsf\_Minimized, wsf\_Restored, wsf\_Maximized. When \text{TaskWinInitFlag} = wsf\_Invisible the loaded application starts in the background mode, otherwise in the foreground mode. The predicate returns the application identifier \text{ApplicationId} of the loaded application (when \text{ReceiveMsgWin} \neq \text{NullWin}). This application identifier will be returned in the \text{e\_EndApplication}(\text{ApplicationId}) event that will be sent to the specified \text{ReceiveMsgWin} to notify about the loaded application termination.

It is possible to terminate previously loaded external application with the application identifier \text{ApplicationId} by the call to \text{vpi\_CloseProcess}:

\[
\text{vpi\_CloseProcess}(\text{ApplicationId})
\]

Notice that as a module loaded by the \text{vpi\_CreateProcess} can be not only an application of the underlying OS, so the call of \text{vpi\_CloseProcess} does not guarantee the specified application termination, for example, if it is an MS-DOS application loaded from a Windows program. Therefore the programmer must wait for \text{e\_EndApplication}(\text{ApplicationId}) event to ensure that the application \text{ApplicationId} has been terminated.

If you specify \text{ReceiveMsgWin} = \text{NullWin}, where \text{NullWin} is a special window handle that should be defined in your program by an expression like the following:

\[
\text{NullWin} = \text{cast} (\text{WINDOW, 0})
\]

then:
• Neither window will receive the `e_EndApplication(ApplicationId)` event to notify that the loaded application has terminated.

• `vpi_CreateProcess` returns `ApplicationId = -1` instead of the real application identifier of the loaded module.

• The loaded application cannot be closed by the call `vpi_CloseProcess(ApplicationId)`.

Notice that the application identifier is not adequate to the application instance handle that can be received as a value of `attr_win_instance` by a call to `vpi_GetAttrVal`.

You should know, that under some platforms it is only possible to have a few simultaneous DC open at the same time. Under Windows 3.1 this number is only 4. Visual Prolog uses an advanced mechanism for caching open DC’s; this is the reason for the ReleaseFlag.
PART 2

The VPI Packages
CHAPTER 28

Dialog Package

Visual Prolog includes a dialog package which makes it easier to handle dialogs. Using the dialog package is optional both on a project basis and on an individual dialog basis - you can create dialogs in your applications entirely without it. If you do wish to make use of it in a project, check the box Project | Application Expert | Options | VPI Options | Dialog Package. This causes the proper files to be included in the project.

The Dialog Package is written entirely in Prolog source and is defined in the files DIALOG.PRO, DIALOG.PRE and DIALOG.DOM. All the global predicates in the Dialog Package are prefixed by 'dialog_'.

To support the dialog package a special tool called the Dialog Package Expert can be used. To activate the Dialog Package Expert for a particular dialog, the Dialog and Window Expert must first be invoked with that dialog, and then the Code Style list box selection set to dialog_Create(...). The Dialog Package Expert can then be invoked by pressing the Dialog Pack push button.

The benefits of using the dialog package are:

- Easy initialization of controls when the dialog is created.
- Easy retrieval of values when the dialog is terminated.
- Automatic handling of the escape key.
- Various checks and validations like checking for mandatory input fields, range checking, default values for fields that were not input, upper- or lower-casing an input field, checking that the length of an input string does not exceed a given value, conversion from the control's value to integer, long, or real, and so on.
- Automatic handling of radio button groups on the platforms that do not support the Auto Button style.
Creating and Initializing a New Dialog

When using the Dialog and Window Expert, it will for a dialog generate the usual `dlg_<name of dialog>_create` predicate, which can create the dialog with the specified layout:

```prolog
dlg_mymodal_Create(Parent):-
    CHECKBOX = b_false,
    AGE = void,
%MARK mymodal, new variables
    dialog_CreateModal(Parent,dlg_mymodal_ResID,"",
%BEGIN mymodal, ControlList, 20:45:09-5.9.1995, Code automatically updated!
        df(idc_checkbox,checkbox(CHECKBOX),nopr),
        df(idc_age,editint(AGE,[mandatory,range(5,99)]),
            str_prompt("Wrong !")),
%END mymodal, ControlList
    dlg_mymodal_eh,0,VALLIST,ANSWER),
    ANSWER = idc_ok,
    dlg_mymodal_update(VALLIST),!.
```

The predicate for creating the dialog will use two different calls from the dialog package depending on whether a modal or a modeless dialog is to be created:

```prolog
WINDOW dialog_CreateModal(WINDOW ParentWindow,RESID ResID, STRING Title, 
    DIALOG_FIELD_LIST InitialFields,EHANDLER, LONG CreationData, 
    DIALOG_VAL_LIST OutputValues,INTEGER ReturnButton)

WINDOW dialog_CreateModeless(WINDOW ParentWindow,RESID ResID, 
    STRING Title,DIALOG_FIELD_LIST InitialFields, 
    EHANDLER,LONG CreationData)
```

The difference is that the call to `dialog_CreateModal` will not return before the dialog is terminated. In this case the return values will be delivered in the two variables `OutputValues` and `ReturnButton`.

When calling `dialog_CreateModeless`, the predicate will return as soon as the dialog is created, while the dialog stays on the screen like a normal window. Collecting any return values from a modeless dialog is normally done when the **OK** button is pressed (or any other button which is used to terminate the dialog).

The argument `DIALOG_FIELD_LIST InitialFields` contains a description of the controls to the dialog package. It is a list of functors, each with the form:
df(ControlId, Properties, ErrorHandling)

Where ControlId is the integer constant that identifies the control within the dialog and ErrorHandling is the action to take if any of the validation actions fail.

The argument DIALOG_VAL_LIST OutputValues for dialog_CreateModal returns in a list the resulting values for the controls when a dialog is terminated. Each element in the list has the form:

dv(DIALOG_CONTROL_ID, DIALOG_ITEM_VAL)

The DIALOG_ITEM_VAL is different for each control type. A predicate is generated to access these resulting values. This is described below.

Note how the Dialog Package Expert inserts new variables. At the mark "MARK mymodal, new variables", new variable bindings will be inserted when you specify new in the Dialog Package Expert. These variable bindings will not be changed/deleted when you make any changes in the Dialog Editor or in the Dialog Package Expert - it is up to you to modify, maintain and delete these.

The Event-handler for a New Dialog

For many dialogs you need not worry about the Event Handler when you use the dialog package. Default handling for the controls are done by the dialog package, and in many cases this suffices.

However, the Dialog and Window Expert will generate the usual Event Handler callback predicate, where it is possible to react to pushbuttons, changed selection in a list box etc. The Dialog and Window Expert can be used to add and edit clauses for the dialog events.

If you are inserting any event handling clauses, you should remember the common rule: If the predicate fails, the dialog package will do the default handling for the event. If the predicate succeeds for an event, no further processing will be done by the dialog package for this event.

Getting the Return Values from a Dialog

The Dialog and Window Expert will generate a predicate for the dialog. The predicate will be called dlg_<dlgname>_update, and it can be used for access to the control values when the dialog is terminated. When the dialog is terminated the dialog package will collect all the values in a list. To retrieve the actual
values, a set of `dialog_VLGet*` predicates can be used. (In this case VL means Value List).

```prolog
dlg_mymodal_update(_VALLIST):-!,
%BEGIN mymodal, Update controls, 20:45:09-5.9.1995, Code automatically updated!
  _CHECKBOX = dialog_VLGetCheck(idc_checkbox,_VALLIST),
  _AGE = dialog_VLGetint(idc_age,_VALLIST),
%END mymodal, Update controls
!.
```

Inside the block of code that is automatically updated, the variables specified in the Dialog Package Expert will be bound to the values of the controls.

It is fairly easy to change the `dlg_<dlgname>_update` predicate so that it returns the values needed in the actual application. But remember, that only the code inside the blocks marked with 'Code automatically updated!', will be changed by the Code Experts; thus you should modify the rest of the generated default code.

### Dialog Domains

You must take a look at the file DIALOG.DOM in the VPI\INCLUDE directory to see the actual defined domains. One thing you should be aware of is that the Dialog Package has a feature to allow working with void values (no value) for the numeric domains. In effect, for the numeric domains, the Dialog Package does not use the values directly, but surrounds them by a functor so it is possible to see if the value is void or not.

```prolog
DIALOG_INT = void; i(INTEGER)
DIALOG_REAL = void; r(REAL)
DIALOG_LONG = void; l(LONG)
```

### Change Enable and/or Show States for Controls

With the predicate `dialog_SetState` it is possible to enable/disable or hide/show a whole list of controls in one call. This feature is powerful and should be used often.

```prolog
dialog_SetState(WINDOW, DIALOG_ENABLE_LIST)
```
Dialog Enable Domains

DIALOG_ENABLE_LIST = DIALOG_ENABLE*
DIALOG_ENABLE =
    enable(DIALOG_CONTROL_ID, BOOLEAN);
    show(DIALOG_CONTROL_ID, BOOLEAN)

Fetch Values of Individual Controls from a Values List.

There are a number of predicates, which can be used to pick out the values from the Value List returned when a dialog is terminated. Each predicate takes the control identifier and the value list as input parameters, and returns the corresponding value:

BOOLEAN dialog_VLGetCheck(DIALOG_CONTROL_ID, DIALOG_VAL_LIST)
DIALOG_CONTROL_ID dialog_VLGetRadiobutton(
    DIALOG_CONTROL_ID, DIALOG_VAL_LIST)
STRING dialog_VLGetStr(DIALOG_CONTROL_ID, DIALOG_VAL_LIST)
DIALOG_INT dialog_VLGetInt(DIALOG_CONTROL_ID, DIALOG_VAL_LIST)
DIALOG_LONG dialog_VLGetLong(DIALOG_CONTROL_ID, DIALOG_VAL_LIST)
DIALOG_REAL dialog_VLGetReal(DIALOG_CONTROL_ID, DIALOG_VAL_LIST)
STRING dialog_VLGetListEdit(DIALOG_CONTROL_ID, DIALOG_VAL_LIST)
INTEGER dialog_VLGetScrollBar(DIALOG_CONTROL_ID, DIALOG_VAL_LIST)
dialog_VLGetListBox(DIALOG_CONTROL_ID, DIALOG_VAL_LIST, SLIST, ILIST)
dialog_VLGetListButton(DIALOG_CONTROL_ID, DIALOG_VAL_LIST, STRING, INTEGER)
dialog_VLGetCustom(DIALOG_CONTROL_ID, DIALOG_VAL_LIST, STRING, LONG)
STRING dialog_VLGetDataField(DIALOG_CONTROL_ID, DIALOG_VAL_LIST)

Set/Fetch Values of Individual Controls in a Dialog

Some dialogs provide a number of actions while the dialog is open. These may be triggered by the activation of a push button at which time other control values may change, or they can be triggered by the selection in a list box, upon which some other values will change. To make this easy, a range of predicates that take the dialog handle and the control identifier as parameters can be used:
Retrieving Values

```c
BOOLEAN dialog_GetCheck(WINDOW, DIALOG_CONTROL_ID)
DIALOG_CONTROL_ID dialog_GetRadiobutton(
    WINDOW, DIALOG_CONTROL_ID FirstFromList)
STRING dialog_GetStr(WINDOW, DIALOG_CONTROL_ID)
DIALOG_INT dialog_GetInt(WINDOW, DIALOG_CONTROL_ID)
DIALOG_LONG dialog_GetLong(WINDOW, DIALOG_CONTROL_ID)
DIALOG_REAL dialog_GetReal(WINDOW, DIALOG_CONTROL_ID)
STRING dialog_GetListEdit(WINDOW, DIALOG_CONTROL_ID)
INTEGER dialog_GetScrollBar(WINDOW, DIALOG_CONTROL_ID)
dialog_GetListBox(WINDOW, DIALOG_CONTROL_ID, SLIST, ILIST)
dialog_GetListButton(WINDOW, DIALOG_CONTROL_ID, STRING, INTEGER)
dialog_GetCustom(WINDOW, DIALOG_CONTROL_ID, STRING, LONG)
STRING dialog_GetDataField(WINDOW, DIALOG_CONTROL_ID)
```

Setting New Values

```c
dialog_SetCheck(WINDOW, DIALOG_CONTROL_ID, BOOLEAN)
dialog_SetRadiobutton(WINDOW, DIALOG_CONTROL_ID FirstFromList, 
    DIALOG_CONTROL_ID Pushed)
dialog_SetStr(WINDOW, DIALOG_CONTROL_ID, STRING)
dialog_SetInt(WINDOW, DIALOG_CONTROL_ID, DIALOG_INT)
dialog_SetLong(WINDOW, DIALOG_CONTROL_ID, DIALOG_LONG)
dialog_SetReal(WINDOW, DIALOG_CONTROL_ID, DIALOG_REAL)
dialog_SetListEdit(WINDOW, DIALOG_CONTROL_ID, STRING)
dialog_SetScrollBar(WINDOW, DIALOG_CONTROL_ID, INTEGER)
dialog_SetListBox(WINDOW, DIALOG_CONTROL_ID, ILIST)
dialog_SetListButton(WINDOW, DIALOG_CONTROL_ID, INTEGER)
dialog_SetCustom(WINDOW, DIALOG_CONTROL_ID, STRING, LONG)
dialog_SetDataField(WINDOW, DIALOG_CONTROL_ID, STRING)
dialog_CheckEditProperties(WINDOW, DIALOG_CONTROL_ID)
```
dialog_CheckEditProperties(WINDOW)
dialog_SetControlTitle(WINDOW,DIALOG_CONTROL_ID,STRING Title)
Editor API

It is possible to use the Visual Prolog editor in the application you create. The file EDIT.PRE in the VPI\INCLUDE directory contains declarations of the predicates that are be used to operate on editor windows. The example in VPI\EXAMPLES\EDITOR shows how to handle normal file editors, which supports loading, saving, cut, paste, etc.

Edit Window Creation

There are basically three predicates to create an editor window. These three predicates determine factors like whether the editor should be a regular editor, whether it should support word wrapping, - or support hypertext fields. The predicates will create a new window.

```prolog
WINDOW edit_Create(WINTYPE wType, RCT rct, STRING FileName, 
        MENU menu, WINDOW ParentWin, WSFLAGS WinFlags, 
        FONT font, BOOLEAN ReadOnly, BOOLEAN Indent, 
        STRING InputStr, ULONG InitPos,EHANDLER handler)

WINDOW edit_CreateWrap(WINTYPE wType, RCT rct, STRING FileName, 
        MENU menu, WINDOW ParentWin, WSFLAGS WinFlags, 
        FONT font, BOOLEAN ReadOnly, BOOLEAN Indent, 
        BOOLEAN Wrap, STRING InputStr, ULONG InitPos, 
        EHANDLER handler)

WINDOW edit_CreateHyper( WINTYPE wType, RCT rct, STRING FileName, 
        MENU menu, WINDOW ParentWin, WSFLAGS WinFlags, 
        FONT font, BOOLEAN ReadOnly, BOOLEAN Indent, 
        BOOLEAN Wrap, STRING InputStr, ULONG InitPos, 
        EHANDLER )
```

Also, for handling an editor in a window that is already created, the following predicate can be used:

```prolog
edit_CreateSubClass(WINDOW Win, STRING FileName, 
        FONT font, BOOLEAN ReadOnly, BOOLEAN Indent, 
        BOOLEAN Wrap, STRING InputStr, ULONG InitPos, 
        EHANDLER handler)
```
One typical use of edit_CreateSubClass is when creating custom controls that are based on the editor.

**Retrieving the Current Text from an Editor Window**

The following two predicates allow retrieving the whole text from an editor window, or just a part of the text in an editor window.

```c
STRING edit_GetText(WINDOW)
STRING edit_GetText(WINDOW, UNSIGNED StartPos, UNSIGNED EndPos)
```

**Adding New Text**

There are three predicates to add new text into an editor window. The text can either be pasted at the current position, at a given position, or at the end by the following three predicates:

```c
edit_PasteStr(WINDOW Win, STRING NewStr)
BOOLEAN edit_PasteStr( WINDOW, ULONG Pos, STRING Str)
edit_AppendStr(WINDOW, STRING Str)
```

**Editing Functions**

The group of editing predicates, which are used to either simulate that a given key has been pressed, or to delete a piece of text relative to the current position.

```c
edit_Char( WINDOW, CHAR )
edit_Enter( WINDOW )
edit_Del( WINDOW )
edit_Backspace( WINDOW )
edit_DelLeftWord( WINDOW )
edit_DelRightWord( WINDOW )
edit_DelToStartOfLine( WINDOW )
edit_DelToEndOfLine( WINDOW )
```
Set/Get Current Selection

This group of predicates can be used to set or retrieve the current selection in the editor window:

- BOOLEAN edit_SetSelection(WINDOW Win, ULONG Pos1, ULONG Pos2)
- edit_GetSelection(WINDOW Win, ULONG Pos1, ULONG Pos2)
- edit_SelectCurLine(WINDOW)
- BOOLEAN edit_SelectWord(WINDOW Win)

Return Position Information

The following predicates can be used to report the current position of the editor caret in either an offset or as Line + Column information.

- ULONG edit_GetPos(WINDOW Win)
- edit_GetPos(WINDOW, INTEGER Line, INTEGER Column)
- INTEGER edit_GetNumberOfLines(WINDOW)

Moving the Caret

Predicates from this group are used to move the current position in the editor.

- edit_Left( WINDOW )
- edit_Right( WINDOW )
- edit_Up( WINDOW )
- edit_Down( WINDOW )
- edit_PgUp( WINDOW )
- edit_PgDn( WINDOW )
- edit_WordLeft( WINDOW )
- edit_WordRight( WINDOW )
- edit_LineHome( WINDOW )
- edit_LineEnd( WINDOW )
- edit_FileHome( WINDOW )
- edit_FileEnd( WINDOW )
- BOOLEAN edit_GotoPos(WINDOW Win, ULONG Pos)
- edit_GotoLine(WINDOW, INTEGER Line)
**Undo/Redo**

The Application can use the Undo and Redo facilities by calling the two predicates:

```lisp
edit_Undo(WINDOW)
edit_Redo(WINDOW)
```

The following predicate can return two flags, which indicate whether it is actually possible to perform an undo / redo operation at the current moment:

```lisp
edit_PossibleUnDoReDo(WINDOW, BOOLEAN Undo, BOOLEAN Redo)
```

**Clipboard Handling**

To handle the Clipboard operations, the following predicates can be called to perform the operations:

```lisp
edit_Cut(WINDOW)
edit_Copy(WINDOW)
edit_Paste(WINDOW)
edit_Delete(WINDOW)
```

**Case Reversing**

The editor has a facility to change the casing of the current selection. The following predicates can be used:

```lisp
edit_ReverseCase(WINDOW)
edit_UpperCase(WINDOW)
edit_LowerCase(WINDOW)
```

**Activating Editor Dialogs**

The editor has a number of internal dialogs. These can be invoked by the following calls:
edit_OptionsDlg()
edit_SearchDlg(WINDOW)
edit_SearchAgain(WINDOW) % Position function
edit_ReplaceDlg(WINDOW)
edit_GotoLineDlg(WINDOW)
edit_GotoPosDlg(WINDOW)
edit_SetFontDlg(WINDOW)

Hypertext Links

If you are working with the hypertext editor, the two following predicates can be used to create and delete hypertext links.

   edit_CreateHyperField(WINDOW)
edit_DeleteHyperField(WINDOW)

To understand the hypertext editor, you should take a look at the VPI\EXAMPLES\HYPER. You should also know that a hypertext link in the editor is inserted as the following text sequence:

   <hyper_begin><Highlighted text><hyper_middle><Hidden text><hyper_end>

Where hyper_begin, hyper_middle and hyper_end are special character values defined in EDIT.DOM.

So if you want to insert a new field you should use some code like:

   format(HyperField,"%c%c%c%c",
        hyper_begin,Highlighted,yper_middle,Hidden,hyper_end),
edit_PasteStr(Win,HyperField),

Suspend/Resume Editor Updating

If you need to do many operations on the editor, you can turn updating off while doing these operations. This is done with the following predicates:

   edit_Suspend(WINDOW)
edit_Resume(WINDOW)

Caret Showing/Hiding

With the following predicate, it is possible to hide or show the caret in the editor:
Hide/Show Status Line

With the following predicate, it is possible to hide the top status line in the editor:

\[
\text{edit\_StatusLineShow}(\text{WINDOW}, \text{BOOLEAN \ Hide/Show})
\]

Color Associations

The token coloring in the editor can be set with the following predicates:

\[
\begin{align*}
\text{edit\_SetColoring}(\text{WINDOW}, \text{INTEGER \ TypeOfColoring}) \\
\text{edit\_SetAssociations}(\text{EDIT\_ASSLIST \ Associations}) \\
\text{EDIT\_ASSLIST} &= \text{edit\_GetAssociations}()
\end{align*}
\]

Editing Options

The various settings for the editor can be set with the following predicates:

\[
\begin{align*}
\text{INTEGER \ edit\_GetTabSize()} \\
\text{edit\_SetTabSize}( \text{WINDOW, INTEGER \ TabSize} ) \\
\text{BOOLEAN \ edit\_GetIndent()} \\
\text{edit\_SetIndent}( \text{WINDOW, BOOLEAN \ On/Off} ) \\
\text{BOOLEAN \ edit\_GetInsert()} \\
\text{edit\_SetInsert}( \text{WINDOW, BOOLEAN \ On/Off} ) \\
\%
\text{Return / Set all options} \\
\text{edit\_GetOptions(\text{ILIST \ ReturnedOptionList})} \\
\text{edit\_SetOptions(\text{ILIST \ OptionList})}
\end{align*}
\]
CHAPTER 30

Message Win Package

The message window package provides a tool for handling a dialog of short messages. Furthermore, by defining clauses for the \texttt{IO\_WrStrCon} predicates defined in IODECEL.PRE in the ROOT\include directory, the message window can capture the output from the \texttt{write} and \texttt{writef} predicates.

When enabling the "message window" option in the Application Expert, the header files for the message window will automatically be included in an application, and the message window will be created.

The Message Window Package is defined in the files MESSAGE.PRO and MESSAGE.PRE in the VPI\include directory.

The Message Window Package defines the following predicates:

\begin{verbatim}
msg_create(INTEGER NoOfLines)
msg_close()
msg_SendMsg(STRING)
msg_SetLines(INTEGER NoofLines)
msg_Clear()
msg_Resize(WINDOW Parent)
msg_SetFocus()
\end{verbatim}

- \texttt{msg_create(INTEGER NoOfLines)}: Create a message window with a given number of lines stored.
- \texttt{msg_close()}: Close the message window.
- \texttt{msg_SendMsg(STRING)}: Append a new string.
- \texttt{msg_SetLines(INTEGER NoofLines)}: Change the Max number of lines.
- \texttt{msg_Clear()}: Clean the window.
- \texttt{msg_Resize(WINDOW Parent)}: Parent window has been resized.
- \texttt{msg_SetFocus()}: Set the focus to the message window.
CHAPTER 31

Toolbar Package

Visual Prolog provides an easy to use toolbar package defined in the files TOOLBAR.DOM and TOOLBAR.PRE.

A toolbar is always associated with a window irrespectively of whether it has the style Left, Right, Top, Bottom, Inside or movable.

The layout of a toolbar will normally be handled in the Toolbar editor, and generating code to create the toolbar should be handled using in the Toolbar Expert.

When the toolbar is used to provide toolbar buttons for menu entries, you need not provide extra coding. This is because toolbar buttons send an e_Menu type event to the parent window.

Creation of a Toolbar

The creation of a toolbar is done by the predicate:

WINDOW toolbar_create(TOOLBAR_STYLE, COLOR, WINDOW Parent, TOOLBAR_LIST)

Where TOOLBAR_LIST is a list of toolbar control descriptions for the toolbar.

Resizing of a Toolbar

The size of the toolbar is adjusted when it is created. It is also important to remember, that toolbars must be resized if the parent window is resized. For the e_Size event, The Dialog and Window Expert will insert a call to:

toolbar_resize(WINDOW ParentWin)

Removing a Toolbar

If a toolbar should be removed, the following predicate can be called:

toolbar_remove(WINDOW ToolbarWin)
Changing the Values in a Toolbar

For check boxes, list buttons and text fields, the following two predicates can be used to get and set the values of the controls:

```
toolbar_SetValue(WINDOW TbWin,MENU_TAG,TOOLBAR_VALUE)
toolbar_GetValue(WINDOW TbWin,MENU_TAG,TOOLBAR_VALUE)
```

Where MENU_TAG identifies the control, and TOOLBAR_VALUE belongs to the following domain:

```
TOOLBAR_VALUE =
    ctrl_value(BOOLEAN,BOOLEAN); % (Enabled,Released)
    text_value(STRING); % (Text)
    lbut_value(BOOLEAN,INTEGER); % (Enabled,PositionInList)
    none
```

Creating Radio Buttons in a Toolbar

In the same way as it is possible to have a group of radio button controls in a dialog, it is possible to specify for a toolbar that a list of check boxes belongs to a group, and only one at a time should be pressed. This facility cannot be defined in the Toolbar editor, but must be specified with a call to `toolbar_GroupControls`:

```
toolbar_GroupControls(WINDOW TbWin,INTEGER GroupNo,TOOLBAR_MENU_TAGS)
```
CHAPTER 32

Tree Package

The tree package is a tool to make it easy to display and work with tree diagrams. The tree can be shown in the four different directions. The example VPI\EXAMPLES\TREE illustrates how to use the tree tool.

The Domains in the Tree Tool

To create a tree, you need to build a structure from the TREE domain. To highlight a path in the tree, you can supply a path from the TREE_PATH domain.

CONSTANTS
  tree_DirRight = 0
  tree_DirLeft  = 1
  tree_DirDown  = 2
  tree_DirUp    = 3

GLOBAL DOMAINS
  TREE_PATH    = LONGLIST*
  LONGLIST     = LONG*
  TREE_NODE_MARK = marked ; unmarked
  TREE_SELECTOR = string
  TREE_DIRECTION = INTEGER
  TREE_ARROW_TYPE = INTEGER
  TREE_ARROW_PEN = tree_arrowPen ( TREE_ARROW_TYPE, PEN )
  TREE_ARROW_PEN_LIST = TREE_ARROW_PEN*
  TREE_WIN_INFO = tree_WinInfo ( FONT, TREE_DIRECTION, TREE_ARROW_PEN_LIST );
  tree_WinInfoAdv ( FONT, TREE_DIRECTION, TREE_ARROW_PEN_LIST, SLIST, PNT )
  TREE = tree ( TREE_SELECTOR, TREE_NODE_MARK, TREELIST, TREE_ARROW_TYPE )
  TREELIST = TREE*
The Predicates in the Tree Tool

Creating a Window with a Tree

\[
\text{WINDOW tree\_Create (}
\text{WINTYPE WinType,RCT,STRING Title,MENU Menu,WINDOW Parent,}
\text{WSFLAGS Flags,EHANDLER Eventhandler,LONG CreationData,}
\text{TREE Tree,TREE\_WIN\_INFO TreeWindowInfo,TREE\_PATH Path )}
\]

Creating a Tree in an Already Open Window (Subclass)

\[
\text{tree\_CreateSubclass( WINDOW, TREE, TREE\_WIN\_INFO, TREE\_PATH )}
\]

Return a List of All Nodes

\[
\text{SLIST tree\_getAllNodes( WINDOW )}
\]

Return the Current Selected Node

\[
\text{TREE\_SELECTOR tree\_GetSelectedNode( WINDOW )}
\]

Return the Parent Node for the Current Selected

\[
\text{TREE\_SELECTOR tree\_GetParentNode( WINDOW )}
\]

Return the Node Type for the Current Selected

\[
\text{TREE\_ARROW\_TYPE tree\_GetNodeArrowType( WINDOW )}
\]

Return Settings from the Tree Window

\[
\text{TREE\_WIN\_INFO tree\_GetWinInfo( WINDOW )}
\]

Return Rectangle for the Tree Window

\[
\text{RCT tree\_GetWinRect( WINDOW, WINDOW OutWindow )}
\]

Return Current Search Value - or Fail

\[
\text{TREE\_SELECTOR tree\_GetCurrentSearch ( WINDOW )}
\]

Move Selection to the First Node Matching the Selector

\[
\text{tree\_SetSelectedNode( WINDOW, TREE\_SELECTOR Selector )}
\]
Sets New Direction for the Tree

    tree_SetDirection( WINDOW, INTEGER )

Sets new Font for the Tree

    tree_SetFont( WINDOW, FONT )

Set a New Tree in the Window

    tree_Update( WINDOW, TREE, TREE_PATH )

Explode/Implode Subtree from the Selected Node

    tree_OpenCloseNode( WINDOW )

Select First Node Using the Selector

    tree_SearchFirstNode( WINDOW, TREE_SELECTOR )

Select next node using the chosen Selector

    tree_SearchNodeAgain( WINDOW )

Make Selected Node Visible

    tree_MoveToSelectedNode( WINDOW )

Move Selection from Child Node to the Parent Node

    tree_MoveToParentNode( WINDOW )

Print the Tree

    tree_Print( WINDOW Win, STRING Title )
CHAPTER 33

Ownerdrawing Package

If you are using ownerdrawing, you should take a look at the following files in the VPI\INCLUDE directory: OWNDRAW.DOM, OWNDRAW.PRE and OWNDRAW.PRO, these files can make ownerdrawing easier.
Index

.A
.BMP file, 119
.CON file, 73, 110
.CUR file, 100
.DLL, 62
.DLL files, 64
.EXE file, 62, 81
.H file, 110
.LIB file, 62
.RC file, 110
.RES file, 66, 74, 110, 111
.VPR file, 110

3
32-bit values, 63
3-State check box, 42

A
advdata, 147
advreq, 147
advstart, 146
advstop, 147
Aldus Placeable Metafile, 151
Anisotropic scaling, 94
AnsiToOem characters, 42
AnsiToOem function, 42
API
  MS Windows, 155
API functions, 155
Application Expert, 113
attr_CheckPortability, 126
attr_color_activeborder, 129
attr_color_activecaption, 129
attr_color_appworkspace, 129
attr_color_btnface, 129
attr_color_btnshadow, 129
attr_color_btntext, 129
attr_color_captiontext, 129
attr_color_desktop, 129
attr_color_graytext, 129
attr_color_highlight, 129
attr_color_highlighttext, 129
attr_color_inactiveborder, 129
attr_color_inactivecaption, 130
attr_color_inactivecaptiontext, 130
attr_color_menu, 130
attr_color_menutext, 130
attr_color_scrollbar, 130
attr_color_tooltipback, 130
attr_color_tooltiptext, 130
attr_color_window, 130
attr_color_windowframe, 130
attr_color_windowstatictext, 130
attr_color_windowtext, 130
attr_ctl_button_height, 127
attr_ctl_checkbox_height, 127
attr_ctl_edit_text_height, 127
attr_ctl_horz_sbar_height, 127
attr_ctl_radiobutton_height, 127
attr_ctl_static_text_height, 127
attr_ctl_vert_sbar_width, 127
attr_dblframe_height, 128
attr_dblframe_width, 128
attr_docframe_height, 128
attr_docframe_width, 128
attr_erase_background, 131
attr_frame_height, 128
attr_frame_width, 128
attr_have_color, 126
attr_have_mouse, 126
attr_icon_heigh, 127
cb_GetBin, 124
cb_GetMetafile, 124
cb_GetPicture, 124
attr_icon_width, 127
cb_GetSize, 124
cb_GetString, 124
attr_menu_height, 128
cb_PictureAvailable, 124
attr_native_window, 130
cb_PutBin, 124
cb_PutMetafile, 124
cb_native_window, 130
attr_num_timers, 130
cb_PutPicture, 124
cb_NUM_TIMERS, 126
cb_PutString, 124
cb_native_window, 130
attr_printer_height, 128
cb_StringAvailable, 124
attr_printer_hres, 128
check marks, 115
cb_Printer_height, 128
attr_native_window, 130
class_Create, 65, 132
attr_native_window, 130
attr_menu_height, 128
class_Destroy, 132
attr_menu_height, 128
classSERVER, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
attr_menu_height, 128
client/server architecture, 138, 140
check box, 51
edit, 50
grid, 63
group box, 49
icon, 49
listbox, 53
push buttons, 50
radio button, 52
text, 49
VBX, 64
Creation Data, 14
CR-LF combination, 124
CURSOR domain, 100
cursor predicates, 101
Custom Controls, 62

D
DC, 156
DDE
   Service name, 138
   Topic name, 138
DDE client, 136
DDE Client operations, 138
DDE connection, 140
DDE events, 11
DDE server, 137
DDE system, 136
   client program, 136
   client/server architecture, 138, 140
closing down, 138
initializing, 138
on MS-Windows platform, 136
server program, 137
dde_AbandonTrans, 141
DDE_ADVFLAGS, 146
dde_Connect, 137, 138, 142, 150
dde_ConnectMult, 142
DDE_CONVMULT, 142
DDE_DATA domain, 141, 145
dde_DisConnect, 138
dde_DisConnectMult, 142
dde_EnableEvents, 148
dde_End, 138
dde_Execute, 139
dde_Init, 138
dde_Poke, 140
dde_PostAdvise, 147
dde_ReConnect, 142
dde_RegisterService, 143
dde_Request, 139
dde_StartAdv, 146
dde_StopAdv, 146
ddeadv_AckReq, 146
ddeadv_NoData, 146
DDEML, 136, 142
DDSHARE, 150
default push button, 51
Device Context, 156
Dialog Base Units, 56
dialog boxes, 12, 66
Dialog Enable domains, 166
dialog package, 162
Dialog Package, 51
DIALOG.DOM, 165
dialog_CreateModal, 163
dialog_CreateModeless, 163
DIALOG_FIELD_LIST domain, 163
DIALOG_INT, 165
DIALOG_ITEM_VAL domain, 164
DIALOG_LONG, 165
dialog_SetState, 165
DIALOG_VAL_LIST domain, 164
dialog_VLGet* predicates, 165
DialogBaseUnits, 91
disconnect, 144
dlg_Ask, 67
dlg_ChooseColor, 89
dlg_ChooseColor2, 71
dlg_ChOOSEFONT, 72
dlg_Error, 72
dlG_GetFileName, 68
dlG_GetStr, 69
dlG_ListSelect, 69
dlG_Note, 73
dlG_PrintSetup, 99
dlG_PrintSetup, 70, 96
DLL
import library, 64
domains
BK_MODE, 78
BRUSH, 78
CONTROL_INFO, 35
CURSOR, 100
DDE_DATA, 141, 145
Dialog Enable, 166
DIALOG_FIELD_LIST, 163
DIALOG_ITEM_VAL, 164
DIALOG_VAL_LIST, 164
DRAWTOOLS, 76
E_HANDLER, 13
ERRHANDLER, 153
EVENT, 25
MENU, 113, 115
MENU_TAG, 27, 177
PEN, 77
SCROLLCODE, 60
TOOLBAR_LIST, 176
TOOLBAR_VALUE, 177
VPI binary, 84
WINDEF_LIST, 15
WINTYPE, 14
WINTYPE, 35
dot pitch, 91
Dots Per Inch
number of, 96
draw_FloodFill, 81
draw_Icon, 81
draw_Pixel, 81
draw_Text, 82
DRAWTOOLS domain, 76
Dynamic Data Exchange, 35, 136
events, 11
Dynamic Link Libraries, 62
E
e_Activate, 37
e_Char, 29
e_CloseRequest, 13
e_CloseRequest(), 26
e_Control, 35, 48
e_Create, 11, 14, 25, 75, 115, 116,
132, 133, 138
e_DDE, 34, 137, 141
e_Deactivate, 37
e_Destroy, 11, 17, 26
e_EndApplication, 37, 159
e_EndSession, 11, 26
e_EraseBackground, 31
e_GetFocus, 31
e_HScroll, 30
e_InitMenu, 27, 124
e_KeyDown, 30
e_KeyUp, 30
e_LoseFocus, 31
e_Menu, 27, 117
e_MouseDbl, 29
e_MouseDown, 28, 29
e_MouseMove, 28, 29
e_MouseUp, 28, 29
e_Move, 31
e_Native, 13, 33, 64, 156
e_OwnerDraw, 33
e_OwnerMeasureItem, 13, 33
e_Size, 32
e_Timer, 34, 118
e_Update, 22, 23, 33
e_User, 13, 34
e_VScroll, 30
edit controls, 50
edit_CreateSubClass, 133
EHANDLER domain, 13
English mapping mode, 93
ERRHANDLER domain, 153
EVENT domain, 25
executable file, 111
execute, 144

F
files
 .CON, 73, 110
 .CUR, 100
 .DLL, 64
 .EXE, 62, 81
 .H, 110
 .LIB, 62
 .RC, 110
 .RES, 66, 74, 110, 111
 .VPR, 110
 Windows .BMP, 119
FLAGS list, 63
FLAGS variable, 63
font_Create, 84
font_GetAttrs, 85
font_SetAttrs, 85

G
General User Interface, 13
getfocus, 36
GOAL, 6
Graphic User Interface, 111
grid controls, 63
group box, 49
GUI, 13, 111, 156
GUI system, 92

H
horizontal scroll bar, 59

I
ICON domain alternative, 49
IO_WrStrCon, 175
IODECEL.PRE, 175

L
list button, 58
listbox, 53
listbox predicates, 53
Logical units, 125
LONG domain, 13
losefocus, 36

M
mapping modes
 English, 93
 Metric, 93
table of, 93
 Twips, 93
MDI
 client window, 131
 flag, 113, 130
 mode, 11, 113
MENU domain, 113, 115
menu_Check, 115
 special version, 116
menu_Enable, 116
 special version, 116
menu_GetRes, 115
menu_PopUp, 116
menu_SetText
 special version, 116
menu_SetText, 115
MENU_TAG domain, 27, 177
menu_Update, 116
MESSAGE.PRE, 175
MESSAGE.PRO, 175
metafile, 151
MetafileAvailable, 124
Metric mapping mode, 93
mf_Close, 152
mf_Destroy, 152
mf_Load, 151
mf_Open, 152
mf_Play, 152
mf_Save, 152
modes
  MDI, 11, 113, 130
  SDI, 12
modified, 36
modified(), 131
Multiple Document Interface, 11, 130

N
NETDDE, 136, 149
NETDDE.DLL, 149
NETDDE.EXE, 150
no_extra_info, 36
non-TopMost windows, 10
NullWin, 158
number of Dots Per Inch, 96

O
OEM characters, 42
OPTLINK, 111
OS/2
  control flags, 44
other_event_handler, 21

P
Palette, 87
PEN domain, 77
pict_Close, 119
pict_Destroy, 120
pict_Draw, 120
raster operations, 120
  stretching version, 120
pict_FromBin, 119
pict_GetFromRes, 122
pict_GetFromWin, 121
pictGetSize, 122
pict_Load, 122
pict_Open, 119
pict_Rotate, 122
pict_Save, 122
pict_ToBin, 119
PM
  control flags, 44
PNT structure, 28
poke, 145
predicates
  caret handling, 125
cursor, 101
  listbox, 53
print_AbortJob, 98
print_EndJob, 97
print_EndPage, 98
print_GetConfig, 98
print_SetConfig, 99
print_SetOrientation, 97
print_StartJob(), 97
print_StartPage, 98
project database, 110
Prolog Global Stack, 145
push button
  default, 51
push buttons, 50

R
radio button, 52
raster operations, 121
  table of, 121
RCT structure, 18
RCTdomain, 10
RCTstructure, 90
RCT structures, 10
rect_GetClient, 18, 19
rect_Inflate, 105
rect_IsEmpty, 105
rect_Offset, 19, 106
rect_PntInside, 106
rect_Union, 106
res_menu, 111
RGB color model, 87
RGB triplet, 87

toolbar_remove, 176
toolbar_resize, 176
toolbar_SetValue, 177
TOOLBAR_VALUE domain, 177
Top-level window, 11
trans_complete, 141
trap, 139
tree_CreateSubclass, 179
tree_CreateSubClass, 133
tree_GetAllNodes, 179
Twips mapping mode, 93

S
safety-net for VPI applications, 153
Screen window, 11
scroll, 35
scroll bar
  horizontal, 59
  vertical, 59
SCROLL_TYPE, 61
scrollbar
  positioning, 61
  proportioning, 62
SCROLLCODE domain, 60
SDI mode, 12
STOP_COLOR, 81
str_DOSStr2, 124
subclass, 134

T
Task window, 11
text controls, 49
timer_Kill, 118
timer_Set, 34, 118
TOOLBAR.DOM, 176
TOOLBAR.PRE, 176
toolbar_create, 176
toolbar_GetValue, 177
toolbar_GroupControls, 177
TOOLBAR_LIST, 176

toolbar_remove, 176
toolbar_resize, 176
toolbar_SetValue, 177
TOOLBAR_VALUE domain, 177
Top-level window, 11
trans_complete, 141
trap, 139
tree_CreateSubclass, 179
tree_CreateSubClass, 133
tree_GetAllNodes, 179
Twips mapping mode, 93

V
Value List, 165
VBX controls, 64
VDE Dialog editor, 66
vertical scroll bar, 59
Visual Basic Runtime environment, 64
VPI binary domain, 84
VPI code, 63
VPI layer, 155
VPI.CON, 28, 30, 74, 88
vpi_ArgValidation, 153
vpi_CheckExistDialog, 111
vpi_CheckExistString, 112
vpi_CloseProcess, 158
vpi_ComposeRGB, 89
vpi_CreateProcess, 37, 158
vpi_GetAttrVal, 88, 131
vpi_GetResStr, 112
vpi_GetTaskWin, 21
vpi_HelpClose, 135
vpi_Init, 6, 11, 126, 131
vpi_ProcessEvents, 107
vpi_ProcessEvents/1, 107
vpi_ProcessEvents/2, 108
vpi_SetAttrVal, 6, 131
vpi_ShowHelp, 135
vpi_ShowHelpContext, 135
vpi_ShowHelpKeyWord, 135

W

win_BringToTop, 22
win_CaptureMouse, 103
win_Check, 51
win_Create, 14, 74, 111, 113
win_CreateControl, 66, 74
win_CreateDyn, 16, 91
win_CreateDynControl, 16, 91
win_CreateDynDialog, 16, 91
win_CreateDynDialog(), 74
win_CreateResDialog, 15, 26, 74, 111
win_Destroy, 11, 13, 17, 26
win_DPtoLP, 94
win_EnableHook, 33, 156
win_GetActiveWindow, 22
win_GetAttrVal, 131, 156
win_GetBrush, 79
win_GetClientRect, 17
win_GetClip, 18
win_GetCtlHandle, 39
win_GetData, 21
win_GetDrawMode, 80
win_GetDrawTools, 76
win_GetFocus, 22
win_GetFont, 79, 85
win_GetFontMetrics, 85
win_GetMapScale, 94
win_GetNativeGraphicContext(), 156
win_GetOuterRect, 18, 19
win_GetParent, 21
win_GetPen, 77
win_GetResDialog, 111
win_GetState, 9, 19
win_GetTextExtent, 83
win_GetType, 8
win_Invalidate, 23, 32, 34

win_LPtoDP, 94
win_MapPoints, 90
win_Move, 18, 32
win_NeedsUpdate, 23
win_PostEvent, 108
win_ReleaseMouse, 103
win_SendEvent, 13, 108
win_SetBackColor, 83
win_SetBackColor, 78
win_SetBrush, 79
win_SetClip, 17
win_SetData, 21
win_SetDrawMode, 80
win_SetDrawTools, 76
win_SetFocus, 22
winSetFont, 79, 85
win_SetForeColor, 83
win_SetHandler, 21
Win_SetIcon, 20
win_SetMapMode, 93, 152
win_SetMapScale, 93, 152
win_SetPen, 78
win_SetState, 9, 19, 20
win_SetSubClassHandler, 134
win_SetText, 20
win_Update, 24
win.ValidateRect, 23
WINBIND sub-directory, 155
WINDEF_LIST, 15, 48, 66
windows
Child, 12
Screen, 11
Task, 11
Top-Level, 11
Windows .BMP file, 119
WINTYPE, 8, 14, 35, 74
wsf_DlgBorder, 128
wsf_Invisible, 158
wsf_Maximized, 158
wsf_Minimized, 158
wsf_Restored, 158
WYSIWYG output, 96