XVT Platform-Specific Guide - Win32

xvt

XVT/WIN32

CONTENTS

XVT/WIN32/64

PREFACE

About This Manual

XVT takes pride in its documentation, and continually seeks to improve it. If you find a documentation error, please contact Customer Support. They will forward your suggestion to XVT's documentation team.

Conventions Used in This Manual

In this manual, the following typographic and code conventions indicate different types of information.

General Conventions

code
: This typestyle is used for code and code elements (names of functions, data types and values, attributes, options, flags, events, and so on). It also is used for environment variables and commands.

code bold
: This typestyle is used for elements that you see in the user interface of applications, such as compilers and debuggers. An arrow separates each successive level of selection that you need to make through a series of menus, e.g., **Edit=>Font=>Size**.

bold
: Bold type is used for filenames, directory names, and program names (utilities, compilers, and other executables).

italics
: Italics are used for emphasis and the names of documents.

Tip: This marks the beginning of a procedure having one or more steps. Tips can help you quickly locate “how-to” information.

Note: An italic heading like this marks a standard kind of information: a Note, Caution, Example, Tip, or See Also (cross-reference).

γ *This symbol and typestyle highlight information specific to using XVT-Design, XVT’s visual programming tool and code generator.*

Code Conventions

<non-literal element> or *non_literal_element*
Angle brackets or italics indicate a non-literal element, for which you would type a substitute.

[optional element]
Square brackets indicate an optional element.

...
Ellipses in data values and data types indicate that these values and types are opaque. You should *not* depend upon the actual values and data types that may be defined.

1

INTRODUCTION

Welcome to XVT/Win32/64/. This platform-specific book (PSB) contains information about using the latest release of the XVT Portability Toolkit (XVT/Win32/64) on your particular platform.

Your release media will have instructions for installing XVT/Win32/64. Once you have XVT/Win32/64 installed, XVT recommends that you try the sample programs that come with the product.

Note: Before writing your application, read the *XVT Portability Toolkit Guide*. The *Guide* focuses on strategies for developing portable applications.

See Also: For an alphabetical listing of all XVT functions and other API elements, refer to the *XVT Portability Toolkit Reference*.
For additional information not documented in this platform-specific book, see the **readme** file in the **doc** directory.

1.1. Changes to Existing Features

For this release of XVT/Win32/64, XVT has made the following changes to existing features:

- The compile constant, used for compiling XVT/Win32/64/64/64/64 platform-specific code, NTWS, has been changed to Win32/64WS. (For backward compatibility, NTWS will continue to work.) See section 2.2.1 of this manual and XVTWS in the online
XVT Portability Toolkit Reference for more information.
- XVT/Win32/64 applications may now specify resource Dynamic Link Libraries (DLLs) at execution startup. This feature is useful in internationalizing your application and for selecting locale-sensitive application resources. Refer to section 3 of this manual and to ATTR_RESOURCE_FILENAME in the online *XVT Portability Toolkit Reference* for more information.

See Also: For a complete listing of XVT/Win32/64 non-portable attributes and escape codes, see Appendix A.

1.2. Compilers Supported by XVT/Win32/64

XVT/Win32/64 supports the following compiler:

- Microsoft Visual Studio 7 (i.e. Visual Studio .NET)

Note: In transitioning to this version of the Microsoft Visual Studio tool, we have worked very hard to maintain backward compatibility with Visual Studio 6.

See Also: Changes to compiler support may be listed in the **readme** file in your **doc** directory.

1.3. XVT Implementations and Operating Systems

The XVT library is currently available for several different window systems and operating systems:

XVT Product:	Window Systems:	Operating Systems:
XVT/Mac	Carbon	MacOS 8.6 and above with CarbonLib installed, Mac OS X 10.1 and above
XVT/Win32/ 64/64/64/64	Win32/64/64/64/64	Windows (all),
		Windows (all)
XVT/XM	X and Motif	UNIX

3

DEVELOPMENT ENVIRONMENT

3.1. Introduction

γ *If you are using XVT-Design, you will rarely, if ever, need to deal directly with makefiles, include files, compiler options, libraries and linkers. Unless you need to modify the makefile templates supplied by XVT-Design, you'll only need to refer to the information in this section (3.1).*

This chapter gives detailed information on building XVT/Win32/ 64Win32/64 applications.

3.1.1. Include Files

γ *XVT-Design generates code that automatically includes all necessary header files.*

To build XVT applications, you must include the XVT-specific header file **xvt.h** in addition to any other application-specific header files.

When writing native code, you can define the platform-specific macro XVT_INCL_NATIVE prior to including **xvt.h**. XVT/Win32/64 then includes the Win32/64 header file **windows.h** for you.

Example: This code fragment shows the proper sequence for calling the macro:

```
#define XVT_INCL_NATIVE #include "xvt.h"
```

3.1.2. Libraries

XVT/Win32/64 uses the following types of libraries:

Static libraries
Linked with your application code to produce the application's executable file. They provide the startup functionality for your application's executable and a roster of entry points for accessing the Dynamic Link Libraries. XVT/Win32/64 requires you to compile and link your applications with the static library appropriate to your compiler.

Dynamic Link Libraries (DLLs)
Dynamically bound to your application at runtime. The entry points linked into your application with the static libraries are resolved to function calls into these libraries at runtime. XVT/Win32/64 requires you to compile and link your applications with the static libraries appropriate for your development environment and application.

See Also: For more information about the DLLs that you must ship with your application, see section 3.2.2.2.

3.2. Microsoft Visual Studio Development Environment

γ *If you use XVT-Design, you will rarely, if ever, need to deal directly with compiler options, libraries, and linkers. Read this section only if you need to modify the makefile templates supplied by XVT-Design.*

This section describes how to build XVT/Win32/64/64/64/64 applications on the Intel x86 platform. The information in this section assumes that you already know how to compile and link non-XVT applications on your platform(s). If you don't, see the Microsoft Visual Studio online documentation for more information.

3.2.1. Compiling: Microsoft cl

γ *When generating makefiles, XVT-Design supplies the appropriate compiler options.*

You can compile your XVT/Win32/64 application just as if you were compiling a non-XVT/Win32/64 application. You can use any Microsoft Visual Studio (**cl**) compiler options that are compatible with the $(cvarsmt) and $(cflags) environment variables. These environment variables are pre-defined for the Microsoft **nmake** utility for Win32/64 by including the file **ntWin32/64.mak**.

For the Microsoft Visual Studio compiler, XVT recommends a command line similar to this:

```
cl $(cvarsmt) $(cflags) -I. -I..\include -Fosample sample.c
```

Note: Although the command line shown here is printed on several lines, you should enter a command line as one line.

Compiler Options

$(cvarsmt)
: Microsoft multi-thread options (*recommended*).

$(cflags)
: Microsoft C compile options (*required*).

-D
: Define macro STRICT for strict type checking (*recommended*).

-I
: Include paths (*required*).

-Fo
: Output filename.

Compiler Optimization Flag

XVT provides a compiler optimization flag, XVT_OPT, for runtime optimization of the PTK. This flag is described further in the *XVT Portability Toolkit Guide*. To use the flag with the Microsoft Visual Studio compiler, you must add a define for the XVT_OPT symbol on the compiler line:

```
cl $(cvarsmt) $(cflags) -DXVT_OPT -I. -I..\include -Fosample sample.c
```

Now recompile and link your application.

Note: All callback functions (event hooks, event handlers, etc.) must use the __cdecl linkage convention. XVT recommends using the XVT_CALLCONV1 macro (defined in **xvt_env.h** and set in **xvt_plat.h**) in your prototypes and headers for callback functions.

See Also: For examples of how to use the XVT_CALLCONV1 macro, see the XVT header files and the online *XVT Portability Toolkit Reference*.

3.2.2. Link Libraries

γ *XVT-Design's makefile templates supply a default configuration that links the appropriate libraries automatically.*

3.2.2.1. Static Libraries

XVT/Win32/64 requires you to compile and link your applications with the static libraries appropriate for your development environment and application.

XVT supplies the following static libraries for Microsoft Visual Studio:

Library:	Functionality of Library:
xvtnmapi.lib	Base XVT library
xvtnmhb.lib	Portable bound help viewer library
xvtnmhn.lib	Native standalone help viewer library
xvtnmtes.obj	Text edit stub
xvtnmdll.obj	DLL stub
xvtnmdcr.obj	Dynamically-linked C-runtime object

You must link the base library, **xvtn*api.lib**, with every XVT/Win32/64/64/64/64 application. If your application uses the XVT online help system, you must link one (and only one) of the help static libraries (**xvtn*h*.lib**) with it. (You are not required to link a help library if your application does not use a help viewer.) If your application does *not* use text edit objects, you can link the text edit stub (**xvtn*tes.obj**) to resolve internal references to the text edit DLL.

If you are linking with **msvcrt.lib** (the dynamically-linked C-runtime library), you must also link the new object **xvtn*dcr.obj** (**dcr** for **D**_ll _**C**_**R**_untime). Linking this object corrects external references not exported by the dynamically-linked C-runtime library.

Note: XVT/Win32/64currently supports only the XVT bound help viewer and the native standalone help viewer. At this time, XVT/Win32/64does not support the XVT portable standalone help viewer, **helpview**.

See Also: For use of the DLL stub **xvtn*dll.obj**, see section 3.2.4.

3.2.2.2. Dynamic Link Libraries

XVT provides a version-named copy of each of the DLLs, built with the Microsoft Visual Studio compiler. The version-unique naming of the DLLs allows multiple versions of XVT/Win32/64/64/64/64-based applications to run on the same system.

Library:	Purpose of DLL:
xnmba560.dll	Base DLL
xnmhb560.dll	Portable bound help viewer DLL
xnmhn560.dll	Native standalone help viewer DLL
xnmte560.dll	Text edit DLL

The base DLL, **xn*ba560.dll**, is *required* for all XVT/Win32/64/64/64/64 applications and must be shipped with your application.

Depending upon which help system viewer is used (see section 3.2.2.1), you must ship one of the help viewer DLLs (**xn*h*560.dll**) with your application. Ship only the help viewer DLL that is needed. If your application does not use a help system viewer, you do not need to ship any of the help viewer DLLs.

If your application uses text edit objects and/or the portable help viewer, you must ship the text edit DLL (**xn*te560.dll**) with your application. If you have linked the text edit stub, you don’t have to ship this DLL.

Note: Win32/64 does *not* permit DLLs to be renamed once they are built.

3.2.3. Linking as an Application Executable: Microsoft link

γ *XVT-Design generates the appropriate command line for you.*

Your XVT/Win32/64 applications link in the usual way for Microsoft Visual Studio, using **link**. As it does for compiling, the Microsoft **nmake** utility provides macros for command line options.

Here is a typical makefile command line for an XVT application:

```
$(link) $(guiflags) -out:sample.exe sample.obj sample.res xvtnmapi.lib
$(guilibsmt)
```

Note: Although the command line shown here is printed on several lines, you should enter a command line as a single line.

Link Options

$(link)
: Microsoft linker (*required*).

$(guiflags)
: Microsoft Visual Studio link flags (*required*).

$(guilibsmt)
: Win32/64 multi-thread link libraries (*recommended*).

To build an XVT/Win32/64 application with a help viewer included:

```
$(link) $(guiflags) -out:sample.exe sample.obj sample.res xvtnmhn.lib
xvtnmapi.lib $(guilibsmt)
```

To build an XVT/Win32/64 application that does not require text edit objects:

```
$(link) $(guiflags) -out:sample.exe xvtnmtes.obj sample.obj sample.res
xvtnmhn.lib xvtnmapi.lib $(guilibsmt)
```

To build an XVT/Win32/64 application that uses dynamically-bound C-runtimes:

```
$(link) $(guiflags) -out:sample.exe xvtnmdcr.obj sample.obj sample.res
xvtnmhn.lib xvtnmapi.lib $(guilibsdll)
```

3.2.4. Building as an Application Dynamic Link Library

You can link the XVT/Win32/64 static libraries into your application DLLs using the DLL stub **xvtn*dll.obj**; however, you must call the function xvt_app_create from the application executable (**.exe** file). If your application will be executing under Win32/64's (Intel x86), you will be required to replace the compiler flag $(cvarsmt) in section 3.2.1 with $(cvarsdll) and to replace the link flag $(guilibsmt) in section 3.2.3 with $(guilibsdll).

To link your module for use with the XVT/Win32/64 libraries in an application Dynamic Link Library (DLL):

Use a link line similar to the following:

```
$(link) -DLL -machine:$(CPU) -out:sample.dll sample.exp
        xvtnmdll.obj xvtnmtes.obj sample.obj xvtnmhn.lib
        xvtnmapi.lib $(guilibsdll)
```

where **sample.exp** is generated from building the **sample.lib** import library:

```
lib -machine:$(CPU) -def:sample.def -out:sample.lib
```

3.2.5. Building Application Resource Dynamic Link Libraries

Application Resource Dynamic Link Libraries may be used to bind locale-specific resources to your internationalized applications.

3.2.5.1. Resource Localization

All XVT Portability Toolkits have moved internal English strings to external resource files to accommodate localization. Each platform defines constants for string resource IDs. The default file **uengasc.h** in the **include** directory contains the XVT/Win32/64 English string constants (other language and character code set localizations will be found there as well).

3.2.5.2. Building Localized Resource Files

To build a resource DLL for use with an XVT/Win32/64 application, create an empty DLL with a binding of your localized resources.

Enter a command line similar to the following:

```
link -dll -machine:$(CPU) -noentry -out:sample.dll sample.res
```

3.2.5.3. Using a Resource DLL

A resource DLL may be bound to your XVT/Win32/64 application at startup time by setting the resource filename prior to calling xvt_ app_create:

```
int XVT_CALLCONV1
main (int argc, char *argv[])) {
    ...
    xvt_vobj_set_attr(NULL_WIN,
        ATTR_RESOURCE_FILENAME, (long) “sample.dll”); ...
    xvt_app_create(argc, argv, 0L, task_eh, &config); ...
}
```

3.2.6. Building Utility Programs

XVT/Win32/64 supplies makefiles for the **errscan** utility program.

To build a utility program using the Microsoft Visual Studio compiler:

1. Move to the appropriate subdirectory. For example, if you are using Intel x86:

   ```
   cd \xvtdsc56\w32_x86\src\errscan
   ```

2. Type: nmake /f makemsc
3. To install a utility program, copy the **.exe** to the **bin** directory like this:

   ```
   nmake /f makemsc install
   ```

3.2.7. For Source Customers Only: XVT/ Win32/64/64/64/64 Development Environment

This section contains information pertinent to XVT/Win32/64 source customers. If you have purchased the XVT/Win32/64 binary product, you can skip this section.

3.2.7.1. Makefiles

XVT/Win32/64provides makefiles with the source code for building the XVT/Win32/64 base and text edit libraries, and the help viewer libraries.

Note: XVT/Win32/64 does not currently support the XVT portable standalone help viewer, **helpview**, and does not ship it in binary form. The source code for the portable standalone help viewer is provided to source customers, but will require modification before it can be used.

The makefiles for the XVT/Win32/64 library are compiler-specific. The makefiles for the XVT libraries have variables that identify the compiler and linker to be used.

3.2.7.2. Building Utility Programs

For source customers, XVT/Win32/64 supplies makefiles for the utility programs **xrc** and **helpc**.

To build a utility program using the Microsoft Visual Studio compiler:

1. Move to the appropriate subdirectory. For example, if you are using Intel x86:

   ```
   cd \xvtdsc56\w32_x86\src\xrc
   ```

2. Type: `nmake /f makemsc`
3. To install a utility program, copy the **.exe** to the **bin** directory using the following command:

   ```
   nmake /f makemsc install
   ```

3.2.7.3. Building the XVT/Win32/64/64/64/64 Libraries

Static Libraries

To build the static libraries:

1. Move to the appropriate subdirectory. For example, if you are using Intel x86:

   ```
   cd \xvtdsc56\w32_x86\src\ptk
   ```

2. Run the **nmake** utility:

   ```
   nmake -f makemsc cleanlib nmake -f makemsc lib
   ```

Dynamic Link Libraries

To build the Dynamic Link Libraries:

1. Move to the appropriate subdirectory. For example, if you are using Intel x86:

   ```
   cd \xvtdsc56\w32_x86\src\ptk
   ```

2. Run the **nmake** utility:

   ```
   nmake -f makemsc cleandll nmake -f makemsc dll
   ```

Static Libraries and Dynamic Link Libraries

To build the static libraries and the Dynamic Link Libraries:

1. Move to the appropriate subdirectory. For example, if you are using Intel x86:

   ```
   cd \xvtdsc56\w32_x86\src\ptk
   ```

2. Run the **nmake** utility:

   ```
   nmake -f makemsc clean nmake -f makemsc
   ```

See Also: For more information, see the makefiles and your compiler documentation.

3.3. Compiling Resources

γ

If you use XVT-Design, you will probably never need to deal directly with resource compiler options. XVT-Design and the ***xrc*** *compiler code resources automatically. The information here is provided for reference purposes only.*

3.3.1. Using xrc

γ

XVT-Design can be configured to invoke ***xrc*** *for you, either directly as part of the code generation process or via a generated makefile.*

To compile XVT URL resources with **xrc**:

Use a command line similar to the following:

```
xrc -r rcnt -i..\..\include -dLIBDIR=.\..\..\lib sample.url
```

See Also: For more information about using **xrc**, see the "Resources and URL" chapter in the *XVT Portability Toolkit Guide*.
For a list of **xrc** options, see the online *XVT Portability Toolkit Reference*.

3.3.2. Using the Native Resource Compiler (rc)

The Win32/64 resource compiler **rc** compiles resource scripts into resource files that have an extension of **.res**. The **rc** compiler can also bind a **.res** file to the executable file that the linker makes.

Compile your resource scripts (***.rc** files) with the **rc** command, just as you do for non-XVT applications.

To compile resource scripts:

Use a command line similar to the following:

```
rc -r sample.rc
```

This creates a file named **sample.res** that will be combined into your **.exe** file by the linker. The best way to generate the resource script file is with **xrc**.

Normally, you use **xrc** to compile menus, dialogs, windows, and strings. Its output is in the form of RC input, so you then have to run **rc**. If a resource can't be written in URL, you'll have to code it directly in RC's language, then embed it in your URL script with a #transparent statement.

γ *The XVT-Design tag User_Url lets you add resource definitions to your application resource file.*

See Also: For more information on using **rc**, see the Microsoft Visual Studio online documentation.

3.4. Building Your Application with the Help System

γ *XVT-Design supplies a default configuration in its makefile template that links with the Win32/64 help viewer, Winhelp. If necessary, you can modify this configuration to suit your needs.*

XVT's hypertext online help system requires a help viewer. For XVT/Win32/64, you can bind the bound (portable) viewer to the application. Alternatively, you can run theWin32/64 help viewer, **Winhelp**.

An application should link with only *one* of the two help libraries discussed below.

Note: XVT/Win32/64 currently supports only the XVT bound help viewer and the native standalone help viewer. At this time, XVT/Win32/64 does not support the XVT portable standalone help viewer, **helpview**.

See Also: For information on the help viewers, see the "Hypertext Online Help" chapter in the *XVT Portability Toolkit Guide*.
For information on the portable help compiler command options, refer to the online *XVT Portability Toolkit Reference* .
For information about the DLLs you must ship if your application uses a portable help viewer, see section 3.2.2.2.

3.4.1. Portable Viewer

XVT/Win32/64 provides the XVT portable hypertext help viewer in bound form. You must use XVT help compiler **helpc** to produce XVT-portable binary help files for the help viewer to use.

To compile help text source files for use with a portable viewer:

Use a command line similar to the following:

```
helpc -f xvt -I..\..\include sample.csh
```

To bind the help viewer to your application:

Link with the following library (in addition to the base XVT library):

xvtnmhb.lib	Bound help viewer library for Microsoft Visual Studio (Intel x86)

Caution: If you are providing context-sensitive help from *modal* XVT windows or dialogs, XVT strongly recommends that you use the native **Winhelp** viewer. The bound viewer is a *modeless* window in XVT. Opening a modeless window from a modal object may result in undefined behavior.

3.4.2. Win32/64/64/64/64 Help Viewer

XVT also provides a portable interface to the Win32/64 help viewer, **Winhelp**. The API is the same as if you were to make function calls to the portable XVT help viewer. For this form of XVT-supported help, you must use the XVT help compiler **helpc** to produce Win32/64 help text source files for the native compiler **hcrtf** to compile.

To compile XVT-format help text source files for use in **Winhelp**:

Use command lines similar to the following:

```
helpc -f win -I..\..\include sample.csh hcrtf -x sample.hpj
```

Note: You must use **hcrtf** to compile the Win32/64 help source files produced by **helpc**. Microsoft Visual Studio no longer provides the **hc3*** help compilers that you previously used to compile help files.

Caution:

To use **Winhelp** with your application:

Link your application with the following library (in addition to the base XVT library):

xvtnmhn.lib	XVT to **Winhelp** communication library for Microsoft Visual Studio (Intel x86)

See Also: For more information on using **Winhelp**, see the *Microsoft Win32/64/64/64/64 Programmer's Reference* and the Microsoft Visual Studio online documentation.

3.4.3. Object Click Mode

Object click mode for XVT's hypertext online help system is *not* standard look-and-feel for Win32/64. Therefore, XVT/Win32/64 does not automatically provide an application menu item which enables this feature for users. If your XVT/Win32/64 application requires context sensitive help using object click mode, you must define the macro, XVT_HELP_OBJCLICK, for compiling URL resources. When this macro is defined, XVT/Win32/64 xrc includes the resource for the **Object Click Mode** menu item in the standard menu. Use a command line similar to the following for compiling URL resources:

```
xrcxrc -r rcnt -i..\..\include
-dLIBDIR=.\..\..\lib -dXVT_HELP_OBJCLICK sample.url
```

Note: Although the command shown here is printed on several lines, you should enter a command line as a single line.

Alternatively, you may define the macro prior to including **url.h** in your application URL resources:

```
#define XVT_HELP_OBJCLICK #include "url.h"
```

2

USING XVT/WIN32/64

2.1. Introduction

This chapter addresses various platform-specific issues that you may need to consider while using XVT/Win32/64. The information here assumes you are familiar with developing Win32/64 applications from a general standpoint. If not, see your compiler's documentation for more information.

2.2. Extensibility

2.2.1. Conditional Compilation

If, in your application, you need to provide some native-platform GUI functionality not available in the XVT Portability Toolkit, then the small percentage of your code that provides that functionality will be non-portable. In this case, you must include the native header file **windows.h**. You must also compile your code conditionally, based on the compilers, the window systems, file systems, and the operating system on which your non-portable code will run.

The XVT Portability Toolkit automatically determines the environment in which the application is running. The XVT header files automatically sense the environment-specific variables that are set implicitly by the compiler.

See Also: For more information on a platform-specific macro, XVT_INCL_NATIVE, that you can call to simplify the process of writing native code, see section 3.1.1.
For more information on environment variables, see the "Symbols for Conditional Compilation" section in the "About the XVT API" chapter of the *XVT Portability Toolkit Guide*, and the file **xvt_env.h** in the **include** directory.

Tip: It's best to consolidate the non-portable code into a few separate files so that most of your application is portable XVT code. Separating your non-portable code makes it easier to change your program when the capability you need is added to a future version of XVT.

Tip: To compile Win32/64-specific code conditionally:

Use the following preprocessor statements to compile window-system-specific code:

```
#if XVTWS == Win32/64WS
/* window-system-specific code goes here */ #endif
```

Use the following preprocessor statements to compile file-system-specific code:

```
#if XVT_FILESYS_NTFS || \
      XVT_FILESYS_DOS || \ XVT_FILESYS_HPFS
/* file-system-specific code goes here */ #endif
```

2.2.2. Multiple Document Interface (MDI)

XVT recommends that you use the Multiple Document Interface (MDI) by setting the attribute ATTR_WIN_MDI. This interface standard specifies a way to design and write application programs that require multiple document windows in Win32/64.

The advantage of MDI is that you can enclose document windows within what is called a "task window". This gives your application a uniform look-and-feel for MS-Windows. The task window has a titlebar, a menu, a sizing border, a system menu icon, a maximize icon, and a minimize icon. MDI also lets you hierarchically enclose document windows and icons.

XVT-Design provides a special tag, Main_Code, that lets you supply code in the Action Code Editor (ACE) before calling xvt_app_create. This enables you to set the ATTR_WIN_MDI attribute before the XVT library assumes control.

γ

See Also: For more information, see the *Microsoft Win32/64/64/64/64/64 Programmer's Reference*. Also, see the description of the ATTR_
WIN_MDI attribute
in Appendix A.

2.2.3. Accessing Window Device Contexts and Handles

Given an XVT WINDOW, your application can access its native window handle (HWND), bitmap handle (HBITMAP), or window device context handle (HDC).

Tip: To get the Win32/64 HWND associated with an XVT WINDOW (excluding windows of type W_PIXMAP, W_PRINT and W_SCREEN):

Call:
```
(HWND)xvt_vobj_get_attr(win, ATTR_NATIVE_WINDOW)
```

Tip: To get the Win32/64 HDC associated with an XVT WINDOW (includes only *drawable* windows of type W_DOC, W_PLAIN, W_DBL, W_MODAL, W_PIXMAP, W_TASK, and W_NO_BORDER):

Call:
```
(HDC)xvt_vobj_get_attr(win,
                              ATTR_NATIVE_GRAPHIC_CONTEXT)
```

Note: You should obtain the ATTR_NATIVE_GRAPHIC_CONTEXT attribute before every use, since the device context is not permanently associated with a WINDOW, and XVT can overwrite it at any time. Also, it is not necessary to delete any retrieved HDC—XVT/Win32/64 does this automatically.

2.3. Invoking an Input Method Editor

The XVT Portability Toolkit is fully compatible with the native Win32/64 Input Method Editor.

An Input Method Editor (IME) is provided by multibyte versions of Win32/64 to allow application users to enter multibyte or other non-ASCII characters from a keyboard that does not support these characters.

On Win32/64, users may invoke the IME by typing Alt + ‘ (Alt key depressed simultaneously with the backquote key). The IME appears as a special entry field at the bottom of the screen except for controls that support IMEs directly (such as text entries and list edits). In IMEs that use composed characters (for example in the conversion from Katakana to Kanji), the space bar is used to perform the composition of selected characters. The IME is disabled when the user is finished entering characters (Enter key) or by toggling with Alt + ‘.

Character events are sent at appropriate times as determined by the IME. If the user composes a character, a character event is sent only *after* the conversion—only the composed character is sent in the event. This means that there may be a delay between when characters are typed and when your application receives an event. Several characters may be typed before any character event is received.

2.4. XVT/Win32/64/64/64/64/64 Resource Specifics

γ

If you use XVT-Design or XVT-Architect, you probably won't need to code native resources directly. XVT-Design, XVT-Architect, and the ***xrc*** *compiler code resources automatically. The information here is provided for reference purposes only.*

This section tells you how to code Win32/64-specific resources. Your Win32/64 resources will be correct if you follow the URL coding guidelines in the "Resources and URL" chapter in the *XVT Portability Toolkit Guide*, because the **xrc** compiler generates native code that adheres to these guidelines.

You can code all menus, dialogs, windows, and strings in URL. Or, you can create them directly in RC. If you must code them in RC, study the output of **xrc** (in RC format) for the resources in the XVT Example Set (**..\samples\design**) to see how they are coded.

Caution: With some Win32/64 controls, you might encounter native (not XVT-imposed) memory limitations. These limitations apply either to individual controls or to all controls in a class of controls. For example, the number of items you can add to a list box, list button, or list edit is determined in Win32/64 by the combined size of the items.

Note: Before reading the rest of this section, familiarize yourself with the **rc** compiler documentation in your compiler's documentation.

2.4.1. Module Name and Icon

γ

*XVT-Design, XVT-Architect, and the **xrc** compiler include the resource definitions in this section automatically. In addition, XVT-Design ensures that the application's module name matches the name specified in the XVT_CONFIG structure.*

The module name for your application must be the same as the application name in your XVT_CONFIG structure. This name must also be the definition for a string resource whose ID is STR_ APPNAME:

```
STRINGTABLE DISCARDABLE BEGIN
    STR_APPNAME, "SAMPLE" END
```

You must define your application's icon ID as ICON_RSRC, like this:

```
ICON_RSRC ICON DISCARDABLE SAMPLE.ICO
```

The symbols STR_APPNAME and ICON_RSRC are defined in other headers included in **xvt.h** (**xvt_plat.h**, **xvt_defs.h**, and **url_plat.h**).

You also should use the module name for the base name of your executable file (e.g., **sample**).

Icons are stored in a file with an extension of **.ico**. XVT requires an icon file with the name ***appname*.ico**, where ***appname*** is the name of your application. The base name of the icon file is the same as the definition you've supplied for the symbol APPNAME. For example, if your URL file has these lines:

```
#define APPNAME SAMPLE #define
QAPPNAME "SAMPLE" #include "url.h"
```

you must have an icon in a file named **sample.ico**.

You can have additional icons for inclusion in dialog boxes; they can be drawn in an XVT window with xvt_dwin_draw_icon.

If you're coding in URL and have followed the instructions in the "Resources and URL" chapter in the *XVT Portability Toolkit Guide*, these resources are defined automatically.

Caution: If you don't use the module name consistently, your application may not link or it may be unable to find its icon.

See Also: For more information, see the description of the ATTR_ WIN_PM_CLASS_ICON attribute in Appendix A.

2.4.2. Cursor Resources

γ

XVT-Design automatically generates an URL file for your application that includes the CURSOR_CROSS and CURSOR_HELP resources. The XVT-Design tag User_Url lets you add cursor resource definitions to your application resource file.

XVT/Win32/64/64/64/64/64 provides the files **croshair.cur** and **objhelp.cur**, which define the XVT portable cursors CURSOR_CROSS and CURSOR_HELP (not supported directly by Win32/64), respectively.

If you want these resources in your application, **croshair.cur** and **objhelp.cur** must be referenced by your URL file, which happens automatically if you include **url_plat.h**. If you do not plan to use these resources, you can remove references to them from the cursors section of **url_plat.h**.

You can also add other cursors to your application resource file. They should have the extension **.cur**.

The following code shows how to add cursors to your URL file:

```
#transparent $$$
1234 CURSOR DISCARDABLE sample.cur $$$
```

2.4.3. Menu Resources

γ

XVT-Design produces menu resources for you. Unless you need to use native menu features unsupported by XVT, you do not have to code menu resources natively.

The best way to learn how to code menus is to lay out a sample menubar in XVT-Design, then let XVT-Design invoke ***xrc****.*

XVT-Design allows you to put items on the menubar that generate a command when activated, instead of popping up a menu. XVT-Design will warn you that a menubar created in this way is non-portable.

You can code menus in native RC syntax by using **xrc** to generate a Win32/64 RC file, then studying it.

When coding resources natively, you can add items to the top-level menubar, and have them generate a command when activated (although this isn’t a good idea because it confuses users).

Additionally, you can use native menu features. For instance, you might want to write native menubar resources that allow a label on a menu to be a bitmap instead of text. Or, you might want to define a submenu with multiple columns.

2.4.3.1. Submenu Tag Assignments

A native RC menu resource does not have tags on its submenus. Since XVT uses tags on both submenus and menu items, it will assign a tag if none exists.

When an application calls an XVT function that requires a menubar, XVT/Win32/64 looks first for a menubar that was described in URL. The submenus on these portable menus have the menu tags that were specified in the URL description.

If XVT/Win32/64 fails to find an URL-generated menubar, it attempts to find a native menu resource. If it finds one, XVT/Win32/64 automatically assigns tags to all the submenus.

2.4.3.2. Creating Non-portable Menubars

You create non-portable menubars by writing them in RC.

Tip: To create a non-portable menubar:

1. Write an RC menu resource using the RC syntax. When this resource is loaded at runtime, XVT synthesizes a submenu tag for each submenu in the menu. The submenu tag is assumed to be the tag of the first item in the submenu, minus one. For example, if the first item of a submenu has a tag of 157, then calling xvt_menu_set_item_enabled(win, 156, FALSE) disables that submenu.
2. Embed the entire RC resource in a #transparent URL construct in your URL file. You can either put the RC resource directly in your URL file, or use the #include directive inside the #transparent construct.
3. Make sure that no menubar exists in your URL file with the same ID as the RC menubar. If two menubars have the same ID, XVT/Win32/64 preferentially loads the portable one, and you will never see the non-portable menubar.

2.4.3.3. Using a Non-portable Menubar Resource ID

You can use the resource ID of a non-portable menubar to specify the menu for a window created with either xvt_win_create or xvt_win_create_def. However, you cannot pass this resource ID to xvt_res_get_menu, because this function always looks for a portable menubar resource, and ignores any non-portable menubars.

2.5. XVT's Encapsulated Font Model

2.5.1. Font Terminology

This section uses the following XVT-defined terms to describe XVT's encapsulated font model:

Physical font
A particular *implementation* of a font as installed on the window system on which an application is running.

Logical font
A *description* of a desired physical font, to a degree of specificity ranging from just a typeface family name or size to a complete description that specifies a particular physical font. A logical font has both portable and non-portable attributes. It is identified by an object of type XVT_FNTID.

γ *In XVT-Design, the job of querying the user for a logical font selection is easy and straightforward. Within any event tag that allows connections, you simply specify a connection to the standard Font selection dialog.*

2.5.2. Native Font Descriptors

To specify a particular physical font, your application can use a native font descriptor, which is a string of data fields. You can include this string as a parameter to xvt_font_set_native_desc, or in URL as part of a FONT or FONT_MAP statement.

The native font descriptor string contains the following fields:

- The native window system and version of the XVT encapsulated font model (the current version is "01").
- Platform-specific fields that the XVT Portability Toolkit decodes and uses to uniquely specify a native font. The fields describe specific attributes of a native font. Each field is separated by a slash, "/".

The native font descriptor string, then, has this format:

```
"<system and version>/<field1>/<field2>/<field3>/ ...<fieldn>"
```

See Also: For more information about specifying fonts, see the "Fonts and Text" chapter in the *XVT Portability Toolkit Guide*.

2.5.3. XVT/Win32/64/64/64/64/64 Font Descriptor Version Identifier

For XVT/Win32/64, the font descriptor version identifier format is NT_<vers>. In this release of XVT/Win32/64, the font descriptor version number is "01," so the font descriptor version identifier is NT_01.

2.5.4. XVT/Win32/64/64/64/64/64 Font Fields

For XVT/Win32/64, the native font descriptor string must contain enough information to populate a LOGFONT structure, which is a 14-part font specification. The following table shows the information from the LOGFONT structure used to map a logical font.

lfHeight	Desired height in logical units
lfWidth	Desired average width in logical units
lfEscapement	Angle of text baseline in tenths of degrees
lfOrientation	Single character orientation in tenths of degrees
lfWeight	Desired weight of character
lfItalic	Italicized text
lfUnderline	Underlined text
lfStrikeout	Struck-out text
lfCharSet	Character set (mapping) desired
lfOutPrecision	Desired precision in matching fonts
lfClipPrecision	Desired clipping of font
lfQuality	Desired font quality
lfPitchAndFamily	General font family and font pitch categories
lfFaceName	Desired font face name

For XVT/Win32/64, the native font descriptor string has this structure:

```
"NT_01/<lfHeight>/<lfWidth>/<lfEscapement>/
    <lfOrientation>/<lfWeight>/<lfItalic>/<lfUnderline>/
    <lfStrikeOut>/<lfCharSet>/<lfOutPrecision>/
    <lfClipPrecision>/<lfQuality>/<lfPitchAndFamily>/ <lfFaceName>"
```

Example: This string shows a valid XVT/Win32/64 native font descriptor string:

```
"NT_01/*/0/0/0/FW_BOLD/1/0/0/ANSI_CHARSET/ OUT_
    DEFAULT_PRECIS/CLIP_DEFAULT_PRECIS/ PROOF_
    QUALITY/DEFAULT_PITCH-FF_ROMAN/*"
```

Note: An asterisk (*) in a native font descriptor string indicates a wildcard condition in which any value is acceptable for that particular field. A hyphen (-) in a font descriptor string cues the font mapper to OR the values.

See Also: For detailed descriptions of the LOGFONT fields, see the *Microsoft Win32/64 Programmer's Reference*.

2.6. Window Geometry

In XVT/Win32/64, you can retrieve information about window geometry by querying attributes using xvt_vobj_get_attr, or through xvt_vobj_get_client_rect and xvt_vobj_get_outer_rect. The following example demonstrates how the values obtained from these attributes and functions are related.

Example: This example was generated from an application running with a SuperVGA device driver. Depending on the device driver you are using, your values will differ.

All numbers are in units of pixels. To facilitate comparison with outer rectangles, client rectangles have been converted into parent-relative coordinates using xvt_vobj_translate_points.

Window Frame Attributes

Using xvt_vobj_get_attr, you obtain the following attribute values:

ATTR_CTL_HORZ_SBAR_HEIGHT	2 2
ATTR_CTL_VERT_SBAR_WIDTH	2 1
ATTR_DBLFRAME_HEIGHT	4
ATTR_DBLFRAME_WIDTH	4
ATTR_DOCFRAME_HEIGHT	4
ATTR_DOCFRAME_WIDTH	4
ATTR_FRAME_HEIGHT	1
ATTR_FRAME_WIDTH	1
ATTR_MENU_HEIGHT	26
ATTR_TITLE_HEIGHT	28

W_DOC (Document) Window Without Scrollbars

Querying the dimensions of a simple W_DOC window (without scrollbars) with xvt_vobj_get_client_rect and xvt_vobj_get_outer_rect, you obtain the following values:

	Left:	Top:	Right:	Bottom:
client rectangle	40	40	240	240
outer rectangle	36	9	244	244

On the left and right of the window, the difference between the outer and client rectangles is the offset ATTR_DOCFRAME_WIDTH, shown in the following relationships:

outer.left = client.left - ATTR_DOCFRAME_WIDTH outer.
right = client.right + ATTR_DOCFRAME_WIDTH

W_DOC (Document) Window With Scrollbars

Attaching scrollbars to the window frame using the creation flags WSF_HSCROLL and WSF_VSCROLL produces these values:

	Left:	Top:	Right:	Bottom:
client rectangle	40	40	240	240
outer rectangle	36	9	264	265

Width

On the right, the difference between the client and outer rectangles is 24, although the sum of ATTR_CTL_VERT_SBAR_WIDTH + ATTR_ DOCFRAME_WIDTH is 25. The one-pixel difference occurs because the *scrollbar* has a frame of width ATTR_FRAME_WIDTH and is overlapped by the window's frame on the right side. This overlapping prevents a thick line from appearing between the scrollbar and the window's frame. Thus the correct relationship is as follows:

outer.right = client.right + ATTR_CTL_VERT_SBAR_WIDTH +
ATTR_DOCFRAME_WIDTH - ATTR_FRAME_WIDTH

Height

A similar situation occurs for the window height. The difference on the bottom is 25, although you might expect it to be 26 (the sum of ATTR_CTL_HORZ_SBAR_HEIGHT + ATTR_DOCFRAME_HEIGHT). In this case, the scrollbar's frame is overlapped by the window's frame at the bottom of the window. The relationship is as follows:

```
outer.bottom = client.bottom + ATTR_CTL_HORZ_SBAR_HEIGHT +
        ATTR_DOCFRAME_HEIGHT - ATTR_FRAME_HEIGHT
```

On the top, the difference is 31, although the sum of ATTR_TITLE_HEIGHT + ATTR_DOCFRAME_HEIGHT is 32. As with scrollbars, the *titlebar* has a frame of height ATTR_FRAME_HEIGHT, which is overlapped by the window's frame at the top.

The relationship is as follows:

```
outer.top = client.top - ATTR_TITLE_HEIGHT -
        ATTR_DOCFRAME_HEIGHT + ATTR_FRAME_HEIGHT
```

W_DBL (Double-bordered) Window

Modal dialogs and W_DBL windows have double borders. The values for a W_DBL window are as follows:

	Left:	**Top:**	**Right:**	**Bottom:**
client rectangle	40	40	240	240
outer rectangle	35	35	245	245

Although ATTR_DBLFRAME_WIDTH is 4, the actual difference between the client and outer rectangles on the left and right sides is 5. This difference occurs because modal dialogs and W_DBL windows have a regular frame outside the double border. (You can see this when you drag a dialog by its titlebar; the outline that follows the movement of the pointer has the dimensions of the regular frame.) Thus the correct relationships are:

```
outer.left =
        client.left - ATTR_DBLFRAME_WIDTH - ATTR_FRAME_WIDTH

outer.right =
        client.right + ATTR_DBLFRAME_WIDTH + ATTR_FRAME_WIDTH

outer.top =
        client.top - ATTR_DBLFRAME_HEIGHT - ATTR_FRAME_HEIGHT
```

```
outer.bottom =
    client.bottom + ATTR_DBLFRAME_HEIGHT + ATTR_FRAME_HEIGHT
```

Window with a Menubar

The last case is a window having a menubar. This could be either TASK_WIN, or a W_DOC window whose parent is the SCREEN_WIN. The values for a W_DOC window are as follows:

	Left:	Top:	Right:	Bottom:
client rectangle	500	100	800	3 0 0
outer rectangle	496	42	824	325

The difference between the client and outer rectangles on the top takes into account the frame, titlebar, and menubar. This difference is 58, although the expected value is 57 (ATTR_DOCFRAME_HEIGHT + ATTR_TITLE_HEIGHT - ATTR_FRAME_HEIGHT + ATTR_MENU_HEIGHT). The ATTR_FRAME_HEIGHT border, which appears between the menubar and the client area, creates the one-pixel discrepancy. Unlike scrollbars and titlebars, ATTR_MENU_HEIGHT does not account for any surrounding borders, so its value must be added when calculating the height of a window having a menubar. The relationship is:

```
outer.top = client.top - ATTR_DOCFRAME_HEIGHT -
    ATTR_TITLE_HEIGHT - ATTR_MENU_HEIGHT
```

2.7. Focus Behavior Using WinExec

In XVT/Win32/64, a call to the Win32/64 function WinExec followed by a call to the XVT function xvt_vobj_destroy can cause an XVT application window to receive focus. However, the window belonging to the task started with WinExec should receive the focus. This section presents two code examples to solve this problem, one for modal dialogs and another for modeless dialogs.

Using XVT-Design, you enter the code for the events in the Action Code Editor (ACE). XVT-Design automatically generates the switch statement along with its cases. The examples in sections 2.7.1 and 2.7.2 show you how you would write the dialog and window handlers yourself.

2.7.1. Sample Code for Modal Dialogs

One solution to the focus problem is to close the window before calling WinExec. The following example illustrates a modal dialog that prompts for the name of a program to run, and then runs it:

```
typedef struct {
        char buf[RESULT_SIZE]; BOOLEAN ok;
} RESULT;


long XVT_CALLCONV1
dlgHandler(WINDOW win, EVENT *ep) {
        RESULT *rp;

        switch (ep->type) { case E_CREATE:?
                rp = (void *)xvt_vobj_get_data(win);
                rp->ok = FALSE;
                rp->buf[0] = 0;
                break;

        case E_CONTROL: switch(ep->v.ctl.id) { case DLG_OK:
                        rp = (void *)xvt_vobj_get_data(win); get_
                        title(xvt_win_get_ctl(win, EDIT_FIELD),
                                        rp->buf, RESULT_SIZE); rp->ok = TRUE;
                case DLG_CANCEL: xvt_vobj_destroy(win); break;
                }
                break;
        }
        return 0;
}
```

```
long XVT_CALLCONV1
winHandler(WINDOW win, EVENT *ep) {
        RESULT res;

        switch (ep->type) {
        case E_MOUSE_DOWN:
                xvt_dlg_create_res(WD_MODAL, RUN_DIALOG, EM_ALL,

                                                        dlgHandler, (long)&res);
                if (res.ok)
                        WinExec(res.buf, SW_SHOW);
                break;
        }
        return 0;
}
```

In this example, the dialog obtains the program name from the user, and records whether the user chose **OK** or **Cancel**. The modal dialog gathers this information and passes it to the calling function so that the calling function can carry out the selection. This is accomplished by passing a pointer to the RESULT structure as the initial application data for the dialog window. If the user chose **OK**, the calling code uses WinExec to run the program. Notice that WinExec is not called in the dialog event handler. This technique works only for a modal dialog, because the struct is filled in before the call to xvt_dlg_create_res returns.

2.7.2. Sample Code for Modeless Dialogs

For a modeless dialog or a window containing controls, the solution to the focus problem is slightly different:

```
        typedef struct {
                char buf[RESULT_SIZE];
                BOOLEAN ok;
                WINDOW notify;
        } RESULT;
```

```
long XVT_CALLCONV1
dlgHandler(WINDOW win, EVENT *ep) {
        RESULT *rp; EVENT e;

        switch (ep->type) { case E_CREATE:
                rp = (void *)xvt_vobj_get_data(win); rp->ok = FALSE;
                break;

        case E_CONTROL:
                switch (ep->v.ctl.id) { case DLG_OK:
                        rp = (void *)xvt_vobj_get_data(win); get_
                        title(xvt_win_get_ctl(win, EDIT_FIELD),
                                rp->buf, RESULT_SIZE); rp->ok = TRUE;
                case DLG_CANCEL: xvt_vobj_destroy(win) break;
                }
                break;
        case E_DESTROY:

                rp = (void *)xvt_vobj_get_data(win); if (rp->ok) {
                        e.type = E_USER; e.v.user.id = DO_EXEC;
                        xvt_win_dispatch_event(rp->notify, &e);
                }
                break;
        }
        return 0;
}
long XVT_CALLCONV1
winHandler(WINDOW win, EVENT *ep) {
        static RESULT res;

        switch (ep->type) {
        case E_MOUSE_DOWN:
                res.notify = win;
                xvt_dlg_create_res(WD_MODELESS, RUN_DIALOG,
                                        EM_ALL, dlgHandler, (long)&res);
                break;
        case E_USER:
                if (ep->v.user.id == DO_EXEC) {
                        WinExec(res.buf, SW_SHOW);
                }
                break;
        }
        return 0;
}
```

Again, the dialog event handler obtains the program name from the user and records whether the user chose **OK** or **Cancel**. Then, when the dialog event handler receives an E_DESTROY event, it uses xvt_win_dispatch_event to notify the window that created the dialog. If the user chose **OK**, the calling function uses WinExec to run the program.

Notice that WinExec is not called in the dialog event handler, and the dialog does not actually perform any action. This ensures that the dialog goes away before WinExec is called. (In the dialog event handler, it is acceptable to have the EVENT on the stack because xvt_win_dispatch_event is just a function call to the window event handler—it doesn't queue an EVENT.)

2.7.3. Calling WinExec with Dynamic Data Exchange (DDE)

In some cases, DDE and WinExec are used together. DDE messages are not XVT events. To handle DDE messages in XVT, you must use an event hook function. First, write the event hook function to process the non-portable DDE messages. Then register the function with XVT using xvt_vobj_set_attr and ATTR_EVENT_HOOK. This event hook function creates corresponding E_USER events and dispatches them to the invisible window's XVT event handler.

Tip: It's generally a good idea to create an invisible, disabled window and register it as the recipient of DDE messages, rather than use one of the visible windows. This technique guarantees that no spurious events will appear for some window as a result of DDE messages, and it isolates all DDE processing to the invisible window's event handler.

If you use some other inter-process communication mechanism on another platform, the only part of the application that needs to change is the event hook function. The introduction of the E_USER layer isolates this change from the rest of the program. Likewise, all processing of the E_USER events is confined to the invisible window's event handler, rather than being interspersed with the event handling for other windows.

2.8. Multi-threaded Applications with XVT/Win32/64/64/64/64/64

XVT/Win32/64 is not thread-safe. However, XVT/Win32/64 builds its libraries with Win32/64 multi-thread capabilities. Using the following guidelines, you can build XVT/Win32/64 applications that are multi-threaded:

- All calls to XVT functions must be in a single thread; XVT strongly recommends you use the main thread.
- To communicate with the thread containing XVT function calls from secondary threads, post (PostMessage) native WM_USER messages to the XVT thread, notifying it that you need to make an XVT call.
- Use native user messages in the range 0–999.

AWin32/64

APPENDIX A: NON-PORTABLE ATTRIBUTES, ESCAPE CODES, AND FUNCTIONS

A.1. Non-portable Attributes

The xvt_vobj_set_attr and xvt_vobj_get_attr functions allow you to manipulate XVT attributes. Non-portable attributes let you fine-tune your application to make it more closely adhere to the look-and-feel of the underlying platform, or to add functionality not provided by the XVT application interface. This section provides a list of non-portable attributes for use with XVT/Win32/64.

See Also: Additional non-portable attributes may be listed in the **readme** file in the **doc** directory.
For detailed interpretation of Win32/64 messages and parameters, see the Microsoft Visual C++ online documentation.

γ *XVT-Design provides a special tag,* SPCL:Main_Code*, that lets you supply code in the Action Code Editor (ACE) before calling* xvt_app_create. *This enables you to set or get system attributes before the XVT library assumes control.*

ATTR_WIN_CMD_LINE

Description: NULL-terminated string containing the command line parameters.

Uses win argument:	No
xvt_vobj_get_attr returns:	NULL-terminated string
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Can use either before or after
Argument type:	STR

ATTR_WIN_CREATEWINDOW_HOOK

Description: A pointer to a function called before the native function CreateWindowEx is called to create a new window. Each parameter is a pointer to a parameter that is passed to CreateWindowEx. Your application can change any parameter before it is passed to the creation function to obtain non-portable window attributes.

Prototype:

```
void XVT_CALLCONV1 createwindow_hook(DWORD * dwExStyle,
        LPCSTR * lpszClassName, LPCSTR * lpszWindowName,
        DWORD * dwStyle, int * x, int * y, int * nWidth,
        int * nHeight, HWND * hwnd, HMENU * hMenu, HINSTANCE *
        hInst, void ** lpvCreateParams)
```

DWORD * dwExStyle
: Native extended window style.

LPCSTR * lpszClassName
: Native class name.

LPCSTR * lpszWindowName
: Native window title.

DWORD * dwStyle
: Native creation style flags.

int * x
: Native x-coordinate (left-side).

int * y
: Native y-coordinate (right-side).

int * nWidth
: Native window width (device units).

int * nHeight
: Native window height (device units).

HWND * hwnd
: Native parent or owner window handle (depending upon style).

HMENU * hMenu
: Native menu or child window handle (depending upon style).

HINSTANCE * hInst
: Native instance data module handle.

void ** lpvCreateParams
: Native application-specific creation parameters.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Installs hook function, or uninstalls hook if value is zero
xvt_app_create use:	Can use either before or after
Default value:	Zero

ATTR_WIN_DELAY_FOCUS_EVENTS

Description: Setting this value to TRUE causes E_FOCUS events to be delayed slightly, so that they are more likely to occur after the native focus events have been processed by the windowing system. This in turn ensures that the application's GUI will be in a more stable state before objects are destroyed, moved, and so forth. The value of this attribute (initially FALSE) can be changed at any time.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Causes E_FOCUS events to be delayed slightly
xvt_app_create use:	Can use either before or after
Default value:	FALSE
Argument type:	BOOLEAN

ATTR_WIN_DRAWABLE_TWIN_BACKGRND

Description: Determines whether a drawable task window is automatically filled with the WINDOW background brush.

Uses win argument:	No
xvt_vobj_get_attr returns:	Current value
xvt_vobj_set_attr effect:	Window background is automatically filled if value is TRUE
xvt_app_create use:	Must use before
Default value:	FALSE

ATTR_WIN_ENHANCED_COMBOBOX

Description: Enables the platform-specific CBS_HASSTRINGS and CBS_OWNERDRAWFIXED attributes. These attributes allow the developer to set font and color properties for individual items in the list. CBS_HASSTRINGS - An owner-draw combo box contains items consisting of strings. The combo box maintains the memory and pointers for the strings so the application can use the GetText member function to retrieve the text for a particular item. CBS_ OWNERDRAWFIXED - The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.

Uses win argument:	No
xvt_vobj_get_attr returns:	Current value
xvt_vobj_set_attr effect:	Window background is automatically filled if value is TRUE
xvt_app_create use:	Must use before
Default value:	FALSE

ATTR_WIN_ENHANCED_LISTBOX

Description: Enables the platform-specific LBS_HASSTRINGS and LBS_OWNERDRAWFIXED attributes. These attributes allow the developer to set font and color properties for individual items in the list. LBS_HASSTRINGS - An owner-draw list box contains items consisting of strings. The list box maintains the memory and pointers for the strings so the application can use the GetText member function to retrieve the text for a particular item. LBS_ OWNERDRAWFIXED - The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.

Uses win argument:	No
xvt_vobj_get_attr returns:	Current value
xvt_vobj_set_attr effect:	Window background is automatically filled if value is TRUE
xvt_app_create use:	Must use before
Default value:	FALSE

ATTR_WIN_FCN_PRINT_INIT

Description: A pointer to a function that is called by xvt_print_create_win after the print window is created, but before printing is started. This is useful in applications needing custom printer initialization.

Prototype:

```
void XVT_CALLCONV1 fcn_print_init(HDC print_DC)
```

HDC print_DC
    Print window device context.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Installs hook function, or uninstalls hook if value is zero
xvt_app_create use:	Must use after
Default value:	Zero

ATTR_WIN_HTML_EVENT_HANDLER

Description: A pointer to the HTML window event handler. The event handler must be derived from IHtmlEvent Handler class to receive events. The IHtmlEventHandler is based on DWebBrowserEvents2, an event sink interface used to receive event notifications from a WebBrowser control or Internet Explorer application (see MSDN).

Uses win argument:	Yes
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Installs the HTML event handler function, or uninstalls function if value is zero
xvt_app_create use:	Must use after
Default value:	Zero

See Also: The attribute ATTR_WIN_NATIVE_HTML_REFERENCE on page A-10.

Example: Below is sample code showing an implementation in C++.

```
========================== derrived_eh.h
==========================

// must define XVT_INCL_NATIVE to get IHtmlEventHandler definition
#define XVT_INCL_NATIVE
#include "xvt.h"

    extern "C"
    {
        void *create_my_eh();
        void destroy_my_eh( void *eh ); }

class CMyHtmlEh : public IHtmlEventHandler {
public:
    CMyHtmlEH();
    virtual ~CMyHtmlEH();

    virtual void eventBeforeNavigate2( IDispatch* pDispatch,
                                BSTR url,
                                void * reserved,
                                BSTR targetFrameName,
                                SAFEARRAY* postData,
                                BSTR headers, VARIANT_BOOL* cancel );

    virtual void eventCommandStateChange( LONG lCommand, BOOL fEnable )
```

```
        {}    // all unneeded events must have empty body .
              .
              .
        };
========================== derrived_eh.cpp
==========================

void *create_my_eh() {
        return new CMyHtmlEH; }

void destroy_my_eh( void *pi ) {
        delete (CMyHtmlEH *)pi; }

void CMyHtmlEh::eventBeforeNavigate2( IDispatch* pDispatch,




                                        BSTR url,
                                        void * reserved,
                                        BSTR targetFrameName,
                                        SAFEARRAY* postData,
                                        BSTR headers, VARIANT_
                                        BOOL* cancel )
{
        OutputDebugStringW( L"MY eventBeforeNavigate2 - " );
        OutputDebugStringW( url );
        OutputDebugStringW( L"\r\n" );
}
==========================
anxvtapp.c
==========================

        ...
        void *create_my_eh();
        void destroy_my_eh( void *eh ); static void * my_htmlEH;
        ...

        E_CREATE:
                my_htmlEH = create_my_eh();
                xvt_vobj_set_attr( ctl, ATTR_WIN_HTML_EVENT_HANDLER,
                                        my_htmlEH );
                break;

        E_DESTROY:

                xvt_vobj_set_attr( ctl, ATTR_WIN_HTML_EVENT_HANDLER,
                                        NULL );
                destroy_my_eh( my_htmlEH ); my_htmlEH = NULL;
                break;

        ...
```

ATTR_WIN_INSTANCE

Description: Instance handle HINSTANCE of the current application.

Uses win argument:	No
xvt_vobj_get_attr returns:	The current application HINST
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Can use either before or after
Argument type:	HINSTANCE

ATTR_WIN_MDI

Description: Determines whether the task window uses Multiple Document Interface (MDI).

Caution: This attribute cannot be used (set to TRUE) when using drawable task windows (ATTR_WIN_PM_DRAWABLE_TWIN set to TRUE).

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Brings application up in MDI mode if TRUE
xvt_app_create use:	Must use before
Default value:	FALSE
Argument type:	BOOLEAN

See Also: The attribute ATTR_WIN_PM_DRAWABLE_TWIN on page A-13.

ATTR_WIN_MDI_CLIENT_HWND

Description: Gets the handle for the MDI client window when ATTR_WIN_MDI has been set.

Uses win argument:	No
xvt_vobj_get_attr returns:	The MDI client window HWND
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Must use after
Argument type:	HWND

ATTR_WIN_MENU_CACHE_COUNT_MAX

Description: An XVT/Win32/64 application can control how Win32/64 resources are used for menus by specifying the maximum number of menus to be retained or cached in a fully expanded, displayable form. Limiting the resources dedicated to menus can substantially increase the number of windows that an application can create. Menus that are not cached are retained in a more compact form, then converted to a displayable form when the associated window is activated.

For larger and more complex menus, you might have to reduce the cache count maximum. Setting the attribute value to zero turns off the caching feature.

Uses win argument:	No
xvt_vobj_get_attr returns:	Current value
xvt_vobj_set_attr effect:	Sets the maximum number of complete menus retained in displayable form
xvt_app_create use:	Can use either before or after
Default value:	7
Argument type:	long

ATTR_WIN_NATIVE_HTML_REFERENCE

Description: Gets the handle to the COM interface used by the native Web browser, IWebBrowser2 (see MSDN).

Note: IWebBrowser2::Release() must be called when finished with the pointer.

See Also: The attribute ATTR_WIN_HTML_EVENT_HANDLER on page A-7.

Uses win argument:	Yes
xvt_vobj_get_attr returns:	Pointer to native Web browser window
xvt_vobj_set_attr effect:	None
xvt_app_create use:	Must use after
Default value:	Zero
Argument type:	IWebBrowser2

Example: Code sample for using the IWebBrowser2 COM object in C.

```
#define CINTERFACE #define COBJMACROS #include <exdisp.h>

....

    IWebBrowser2*pWebBrowser;

    pWebBrowser = (IWebBrowser2*)xvt_vobj_get_attr( ctl,
        ATTR_WIN_NATIVE_HTML_REFERENCE );

  IWebBrowser2_GoSearch( pWebBrowser );

  IWebBrowser2_Release( pWebBrowser ); pWebBrowser = NULL;

....
```

ATTR_WIN_NO_PRINT_THREAD

Description: Provides a method of printing without requiring the application to create a print thread. Normally, XVT forces printing functionality to be centralized into a specific function call (thread). ATTR_WIN_NO_PRINT_THREAD allows xvt_print_create_win to be called without having the function defined. If set to TRUE then print thread is not required.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Toggles requirement of specific print function
xvt_app_create use:	After
Default value:	FALSE
Argument type:	BOOLEAN

ATTR_WIN_OPENFILENAME_HOOK

Description: A pointer to a function that is called before the common open file (GetOpenFileName) or save file (GetSaveFileName) dialog is displayed. You can change the OPENFILENAME structure to control the behavior of the common dialogs.

Prototype: void XVT_CALLCONV1 openfilename_hook (OPENFILENAME * lpofn)

OPENFILENAME * lpofn
    Native filename structure.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Installs hook function, or uninstalls hook if value is zero
xvt_app_create use:	Must use after
Default value:	Zero

ATTR_WIN_PM_CLASS_ICON

Description: Stores an icon resource ID that identifies the icon used to represent a minimized window.

This attribute is valid only for iconizable windows; it must be set immediately before window creation. Setting this attribute sets the icon for all subsequent window creations. You will want to reset this attribute immediately after window creation to avoid accidentally creating other windows with the same icon.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Sets icon ID for next top-level window created
xvt_app_create use:	Must use after
Default value:	Zero
Argument type:	int

ATTR_WIN_PM_DRAWABLE_TWIN

Description: Controls whether the task window is drawable.

Caution: This attribute cannot be used (set to TRUE) when using Multiple Document Interface mode (ATTR_WIN_MDI set to TRUE).

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Makes task window drawable if TRUE, or non-drawable if FALSE
xvt_app_create use:	Must use before
Argument type:	BOOLEAN
Default value:	FALSE

See Also: The attribute ATTR_WIN_MDI on page A-9.

ATTR_WIN_PM_NO_TWIN

Description: Setting to TRUE disables the creation of a physical TASK_WIN so that TASK_WIN maps to the screen. This is recommended for applications with one top-level window.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Controls creation of physical TASK_WIN
xvt_app_create use:	Must use before
Default value:	FALSE
Argument type:	BOOLEAN

ATTR_WIN_PM_SPECIAL_1ST_DOC

Description: Causes the first document window to be expanded to completely fill the task window. This directive overrides the RCT parameter to any of the window creation calls for the first window only.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Sets special “first doc” state
xvt_app_create use:	Can use either before or after
Default value:	FALSE
Argument type:	BOOLEAN

ATTR_WIN_PM_TWIN_FRAME_WINDOW

Description: The WINDOW value for the MDI frame window. Parenting portable (XVT) and native (non-XVT) windows to the MDI frame window is how you create frame-parented objects such as toolbars and status bars and ensure they do not get overlapped by other windows when the application is operating in MDI mode.

Uses win argument:	No
xvt_vobj_get_attr returns:	The window for the MDI frame
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Must use after
Argument type:	WINDOW

ATTR_WIN_PM_TWIN_MARGIN_TOP ATTR_WIN_PM_TWIN_MARGIN_BOTTOM ATTR_WIN_PM_TWIN_MARGIN_LEFT ATTR_WIN_PM_TWIN_MARGIN_RIGHT

Description: Setting these attributes defines a "buffer zone" of pixels inside the decorations of the MDI frame window that will not be included in the MDI client area when the application enables MDI mode. (In MDI mode, all windows parented to the task window are natively parented to the MDI client window.)

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Sets the top or bottom, left or right margin inside MDI frame window
xvt_app_create use:	Either before or after
Default value:	Zero
Argument type:	long

ATTR_WIN_PM_TWIN_STARTUP_DATA

Description: Initial application data for TASK_WIN. You can change it with xvt_vobj_set_data and retrieve it with xvt_vobj_get_data.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Specifies application data for TASK_WIN
xvt_app_create use:	Must use before
Default value:	Zero
Argument type:	long

ATTR_WIN_PM_TWIN_STARTUP_MASK

Description: Initial application mask for TASK_WIN. You can change it with xvt_win_set_event_mask and retrieve it with xvt_win_get_event_mask.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Sets task window event mask
xvt_app_create use:	Must use before
Default value:	EM_ALL
Argument type:	EVENT_MASK

ATTR_WIN_PM_TWIN_STARTUP_RCT

Description: Sets creation rectangle for TASK_WIN, which otherwise uses system defaults.

Uses win argument:	No
xvt_vobj_get_attr returns:	Invalid
xvt_vobj_set_attr effect:	Sets the initial TASK_WIN size and position
xvt_app_create use:	Must use before
Default value:	System default
Argument type:	(RCT*)

ATTR_WIN_PM_TWIN_STARTUP_STYLE

Description: Sets WSF_* flags for TASK_WIN.

Uses win argument:	No
xvt_vobj_get_attr returns:	Invalid
xvt_vobj_set_attr effect:	Sets TASK_WIN style
xvt_app_create use:	Must use before
Default value:	WSF_ICONIZABLE|WSF_SIZE|WSF_CLOSE
Argument type:	long

ATTR_WIN_POPUP_DETACHED

Description: Setting this value to TRUE causes subsequently created windows whose parent is SCREEN_WIN to have a pop-up style instead of an overlapped style.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Sets style of next detached window
xvt_app_create use:	Must use after
Default value:	FALSE
Argument type:	BOOLEAN

ATTR_WIN_PREV_INSTANCE

Description: The handle of the previous instance of the application, if one is running. If no other instances are running, this attribute's value is zero.

Note: XVT/Win32/64/64/64 will always return NULL for this attribute. The attribute is provided to maintain consistency with XVT/Win16.

Uses win argument:	No
xvt_vobj_get_attr returns:	NULL
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Can use either before or after
Argument type:	HINSTANCE

ATTR_WIN_R3_DIALOG_PLACEMENT

Description: This attribute determines whether dialogs are placed according to the rules of XVT Release 3.x (R3) or XVT Release 4.x (R4). In Release 4.0 of XVT/NT, now XVT/Win32/64, dialog placement was normalized to be consistent with other platforms and the way they are documented. As long as ATTR_WIN_R3_DIALOG_PLACEMENT is FALSE, initial dialog creation coordinates are relative to SCREEN_WIN, not TASK_WIN. On the other hand, when this attribute is set to TRUE, to simulate R3 dialog placement, dialogs are initially placed relative to TASK_WIN.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Changes paradigm by which dialogs are placed on the screen
xvt_app_create use:	Can use either before or after
Default value:	FALSE
Argument type:	BOOLEAN

Example: The sample code shown below sets this attribute in the function main so that all dialogs created by the application are affected:

```
int XVT_CALLCONV1
    main XVT_CALLCONV2 ( int argc, char * argv[ ] ) {

    #if ( XVTWS == WINWS ) || ( XVTWS == NTWS )
        xvt_vobj_set_attr( NULL_WIN,
                ATTR_WIN_R3_DIALOG_PLACEMENT, TRUE ); #endif

    xvt_app_create( argc, argv, 0L, task_eh,


                    &xvt_config );
    return 0;
}
```

Note: When set to TRUE, ATTR_WIN_R3_DIALOG_PLACEMENT also disables the centering of all standard dialogs, which was also a characteristic of dialog placement in XVT/NT R3.

ATTR_WIN_RAW_EVENT_HOOK

Description: A pointer to a hook function that is called *before* a native message is translated and dispatched via TranslateMessage and DispatchMessage from the Win32/64 native message queue. The phwnd and pmsg parameters are pointers to data passed internally to Win32/ 64 message procedures.

Your application can process this message data in any appropriate manner. If your hook function returns FALSE, XVT does not translate or dispatch the event. If your hook function returns TRUE, XVT processes the event normally.

Prototype: BOOLEAN XVT_CALLCONV1 raw_event_hook(MSG * pmsg, HWND * phwnd)

MSG * pmsg
    Native message.

HWND * phwnd
    Intended window.

xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Installs hook function, or uninstalls hook if value is zero
xvt_app_create use:	Can use either before or after
Default value:	Zero

ATTR_WIN_TIMER_HI_GRANULARITY

Description: Controls the interval between internal timer events used to process user-requested timer intervals greater than 64K milliseconds. This affects the accuracy with which timer events having long intervals are sent.

xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Sets the granularity
xvt_app_create use:	Must use before
Default value:	1000
Argument type:	long

ATTR_WIN_USE_PCL_RECTS

Description: When XVT/Win32/64 prints to an HP LaserJet printer, it normally uses a special escape provided by Win32/64 to optimize the printing of rectangles with solid or empty interiors. This applies only to HP LaserJet (and possibly Inkjet) printers that use PCL, not to HP LaserJet printers with PostScript extensions.

The Win32/64 HP LaserJet device driver uses two passes to print a page. Normally, the first pass prints text and the second pass prints graphics. The text output is small, since text is encoded by characters, and the graphics output is large, since it is represented by raster images.

If you don’t use this attribute, rectangles, vertical lines, and horizontal lines are output as raster graphics. This approach has three drawbacks:

- It creates potentially enormous spool files
- The HP driver sometimes produces garbage output if too much graphical data is generated
- It is slow

XVT uses the special HP escape to reduce drawing time, reduce spool file size, and improve reliability. However, be aware of the following trade-off. Rectangles drawn with this escape are effectively OR’d with the raster output of other drawing functions, which means that if you draw a rectangle, and then draw an overlapping oval, the rectangle “shows through” the oval.

To disable this behavior, and produce correct but much larger output, disable the attribute (set it to FALSE).

Tip: You can enable or disable this attribute at any time, including during printing. For most programs, XVT recommends that you leave this attribute enabled, as it can greatly reduce output size and increase printing speed. For example, a page of “spreadsheet” output, where cells of text are separated by grid lines, is reduced from about 1300KB to about 60KB.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previous setting
xvt_vobj_set_attr effect:	Sets drawing mode for HP printers
xvt_app_create use:	Must use after
Default value:	TRUE
Argument type:	BOOLEAN

ATTR_WIN_USE_PRINT_BANDING

Description: This attribute controls whether XVT/Win32/64 uses the banding or non-banding native APIs when printing. Note that this attribute can only be set before starting the print job (when the application calls xvt_print_create_win).

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Can use either before or after
Argument type:	BOOLEAN
Default value:	TRUE

ATTR_WIN_WCLASSREG_HOOK

Description: A pointer to a function that is called before every window class registration (i.e., every call to the RegisterClass function). The application can change the WNDCLASS structure.

Prototype: void XVT_CALLCONV1 wclassreg_hook(WNDCLASS * pwc)

WNDCLASS * pwc
    Window class.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Installs hook function, or uninstalls hook if value is zero
xvt_app_create use:	Can use either before or after
Default value:	Zero

Valid Class Names and Corresponding Window Types

Class Name:	Window Type:
mdidoc	MDI document
mditask	MDI frame
task	non-drawable task
drawtask	drawable task
detdoc	detached document
doc	nested document
child	child
dialog	dialog

γ *In XVT-Design, you supply the class hook definition by means of the Obj_Decl tag in the ACE. You then supply the extra code in the main function (the code prior to the call to xvt_app_create) by means of the Main_Code tag in the Action Code Editor (ACE).*

Example: This example shows how to update the icon in a drawable task window using a simple class registration hook:

```
#define TESTNAME "icon example"


void XVT_CALLCONV1 classHook(WNDCLASS far * pwc) {


        if (strncmp(pwc->lpszClassName, "drawtask", 8) == 0) { pwc-
                >hbrBackground = COLOR_APPWORKSPACE+1;
        }
}

void XVT_CALLCONV1
main(int argc, char * argv[ ])
{
        XVT_CONFIG c;

        memset(&c, 0, sizeof(c));
        c.menu_bar_ID = MENU_BAR_RID;
        c.base_appl_name = TESTNAME;
        c.appl_name = TESTNAME; c.taskwin_title = TESTNAME;

        xvt_vobj_set_attr(NULL_WIN,
                                ATTR_WIN_PM_DRAWABLE_TWIN, TRUE);
        xvt_vobj_set_attr(NULL_WIN,
                                ATTR_WIN_WCLASSREG_HOOK, (long)classHook)
        xvt_app_create(argc, argv, 0L, taskHandler, &c);

}
```

The main function does the normal initialization of an XVT_CONFIG struct. It also calls xvt_vobj_set_attr twice; the first call makes the task window drawable, and the second sets up the class registration hook function.

The hook function detects when the drawable task window's class is being registered by checking the class name. XVT appends some additional information to the class name. For this reason, you should use the function strncmp to test just the relevant part of the name.

A.2. Variations on Portable Attributes

These portable attributes have slight variations in meaning in order to support differences between the native Win32/64 platform and other platforms.

ATTR_EVENT_HOOK

Description: A pointer to a hook function that is called whenever a native Win32/ 64 event is received for a window or dialog in your application. The hwnd, msg, wparam, and lparam parameters are copies of data passed internally to Win32/64 message procedures by XVT/Win32/64. The ret parameter is the return value from XVT used in the native message-handling procedure. In almost all cases, your event hook function should set ret to zero.

Your application can process this message data in any appropriate manner—however, modifying this data will have no effect on any default processing by Win32/64. If your hook function returns FALSE, XVT does not process the event further. If your hook function returns TRUE, XVT processes the event normally.

Prototype:

```
BOOLEAN XVT_CALLCONV1 event_hook(HWND hwnd, UINT msg,
    UINT wparam, ULONG lparam, long * ret)
```

HWND hwnd
: Native window.

UINT msg
: Native message.

UINT wparam
: Native word parameter.

ULONG lparam
: Native long parameter.

long * ret
: Native return value.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Installs hook function, or uninstalls hook if value is zero
xvt_app_create use:	Can use either before or after
Default value:	Zero

ATTR_KEY_HOOK

Note: The key hook function supports internationalization and localization.

Multibyte-nonaware Application

If your application uses a single-byte character code set and you have set the value of ATTR_MULTIBYTE_AWARE as FALSE (default), then ATTR_KEY_HOOK behaves as follows:

Description: A pointer to a hook function that is called *after* native WM_KEYDOWN or WM_CHAR events are received and *before* E_CHAR events are sent to your application. The hwnd, msg, wparam, and lparam parameters are copies of data passed internally to XVT/Win32/64 message procedures by Win32/64.

If you need to perform key translation, you must modify data in the ret parameter. XVT uses the ret parameter to construct an E_CHAR event. The s_char struct appears in the char part of the E_CHAR EVENT substructure, and looks like this:

```
struct s_char {
        XVT_WCHAR ch;                   /* wide character */
        BOOLEAN shift;                  /* shift key? */
        BOOLEAN control;                /* ctrl or
                                             option key? */
        BOOLEAN virtual_key;            /* virtual key? */
        unsigned long modifiers;  /* key bit field
                                             modifiers */
        } chr;
```

If your key hook function translates a character to a virtual key, then it should also set the virtual_key field to TRUE. Your application can process this message data in any appropriate manner—however, modifying the hwnd, msg, wparam, and lparam parameters will have no effect on any default processing by Win32/64. If your hook function returns FALSE, XVT does not process the event further and dispatches the E_CHAR event to your application. If your hook function returns TRUE, XVT processes the event normally and may either dispatch the E_CHAR event to your application or discard the event. (The return values have an opposite sense from that of the multibyte version of this attribute.)

Prototype:

```
BOOLEAN XVT_CALLCONV1 key_hook(HWND hwnd, UINT msg,
    UINT wparam, ULONG lparam, struct s_char * ret)
```

HWND hwnd
: Native window.

UINT msg
: Native message.

UINT wparam
: Native word parameter.

ULONG lparam
: Native long parameter.

struct s_char * ret
: XVT-translated key.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Installs hook function, or uninstalls hook if value is zero
xvt_app_create use:	Can use either before or after
Default value:	Zero

Multibyte-aware Application

If your application is multibyte-aware (in other words, you have set the value of ATTR_MULTIBYTE_AWARE as TRUE), then ATTR_KEY_HOOK behaves as follows:

Description: A pointer to a hook function that is called *after* native WM_KEYDOWN or WM_CHAR events are received and *before* E_CHAR events are sent to your application. The hwnd, msg, wparam, and lparam parameters are copies of data passed internally to XVT/Win32/64 message procedures by Win32/64.

If you need to perform key translation, you must modify data in the event parameter (an XVT E_CHAR event).

If your key hook function translates a character to a virtual key, then it should also set the virtual_key field to TRUE. Your application can process this message data in any appropriate manner—however, modifying the hwnd, msg, wparam, and lparam parameters will have no effect on any default processing by Win32/64. If your hook function returns TRUE, XVT does not process the event further and dispatches the E_CHAR event to your application. If your hook function returns

FALSE, XVT processes the event normally and may either dispatch the E_CHAR event to your application or discard the event.

Prototype:
```
BOOLEAN XVT_CALLCONV1 key_hook(HWND hwnd, UINT msg,
        UINT wparam, ULONG lparam, EVENT * event)
```

HWND hwnd
: Native window.

UINT msg
: Native message.

UINT wparam
: Native word parameter.

ULONG lparam
: Native long parameter.

EVENT * event
: XVT event.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Installs hook function, or uninstalls hook if value is zero
xvt_app_create use:	Can use either before or after
Default value:	Zero

ATTR_NATIVE_GRAPHIC_CONTEXT

Description: MS-Windows graphics context (device context handle) of an XVT WINDOW.

Uses win argument:	Yes
xvt_vobj_get_attr returns:	HDC
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Must use after
Default value:	None

ATTR_NATIVE_WINDOW

Description: MS-Windows Window which corresponds to the client area of an XVT WINDOW. Not valid for windows of type W_PIXMAP, W_PRINT and W_SCREEN.

Uses win argument:	Yes
xvt_vobj_get_attr returns:	HWND
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Must use after
Default value:	None

A.3. Non-portable Escape Codes

The xvt_app_escape function enables you to set or get XVT/Win32/64-specific information that you cannot set or get using the xvt_vobj_set_attr or xvt_vobj_get_attr functions. The xvt_app_escape function's escape codes and the associated parameter lists are given below, with a brief explanation of types
and values. The escape code is an integer whose value is defined internally by XVT.

XVT_ESC_WIN_TERMINATE

Description: This escape behaves similarly to xvt_vobj_destroy(TASK_WIN) or, equivalently, xvt_app_destroy, in that it terminates the application and sends an E_DESTROY event to the task event handler. However, this escape differs in two important ways:

- XVT_ESC_WIN_TERMINATE returns to the calling function
- XVT_ESC_WIN_TERMINATE causes xvt_app_create to return to the application main function, which also must return

Use this escape in place of calls to xvt_app_destroy and/or xvt_vobj_destroy(TASK_WIN) only. Be sure that your application is prepared for the call to xvt_app_escape to return, since calls to xvt_app_destroy and/or xvt_vobj_destroy(TASK_WIN) are guaranteed never to return.

Note: Win32/64 handles freeing of resources better than MS-Windows 3.1.
In XVT/Win32/64, this escape exists primarily to maintain consistency with XVT/Win16.

Prototype: xvt_app_escape(XVT_ESC_WIN_TERMINATE)

A.4. Non-portable Functions

xvtwi_hwnd_to_window

Description: This function returns the XVT WINDOW for a given native window handle (HWND). If no XVT WINDOW corresponds to the given HWND, the function returns NULL_WIN.

Prototype: WINDOW XVT_CALLCONV1 xvtwi_hwnd_to_window(HWND hwnd)

HWND hwnd
    Handle to a native window.

Caution: This function is not prototyped in the standard XVT application programmer interface (API). If you use this function, you must provide the prototype as needed.

BWin32/64/

Appendix B: Frequently Asked Questions

Q: *When running the* ***rc*** *compiler on a* ***xrc****-generated .****rc*** *file, I receive errors about not finding a* ***.ico*** *file. Where do I find this file?*

A: This is the Win32/64 icon that will be associated with your application.You can copy **xvtapp.ico** from the **lib** directory to the correct filename.

Alternatively, you can create your own icon by using a native icon editor such as the one that ships with Microsoft Visual C++.

Note that the filenames must match the definition of the symbol APPNAME in your URL file. If the filename is not defined, it defaults to **xvtapp**.

See Also: For more information on application icons, see section 2.4.1.

Q: *When running **rc** on a **xrc**-generated **.rc** file, I receive errors about not finding **croshair.cur**. This file exists in the **lib** directory, but **xrc** insists it must be in **.\ ..\ ..\lib**. How can I change this?*

A: The **xrc** compiler uses the definition of the symbol LIBDIR to direct the **rc** compiler to the necessary cursor (**.cur**) files. If this symbol is not set before including **url.h**, it defaults to **.\ ..\ ..\lib**. You can override this on the **xrc** command line, using the -d option.

Example: This example shows how to invoke **xrc** using this option:

```
xrc -r rcwin -dLIBDIR=c:\myxvtdir\w32_x86\lib sample.url
```

Note: The **rc** compiler requires that all relative paths begin with these characters: **.**
Thus the path **..\ ..\lib** is specified as **.\ ..\ ..\lib**.

See Also: For more information on preprocessor symbol definition, see the "Resources and URL" chapter in the *XVT Portability Toolkit Guide*.

Q: *When I first enter a debugging session of an XVT/Win32/64/64 application, I do not see my source code, only assembly language. Why is this?*

A: The entry point into all Win32/64 applications is a function known as WinMain. For an XVT/Win32/64 application, this is contained in the static library, not in your own application's code. Consequently, what you see is the assembly language for XVT/Win32/64/64's WinMain, the source for which is not included in a binary installation.

You need to make sure that $(cdebug) is in the compile command and that $(ldebug) is in the link command line.

Inside Visual C++, you should view the source code for your application and set breakpoints where you want to start your debugging.

See Also: For more information, see the Microsoft Visual Studio online documentation.

Q: *How do I use color with controls in my application?*

A: You can use the following two Portability Toolkit functions to set colors for controls in your application:

```
void xvt_ctl_set_colors(WINDOW ctl_win, // WINDOW ID of the control
XVT_COLOR_COMPONENT *colors, // colors to set or unset
XVT_COLOR_ACTION action) // set or unset the colors
```

and

```
void xvt_win_set_ctl_colors(WINDOW win,
        // WINDOW ID of the window or dialog
XVT_COLOR_COMPONENT *colors, // colors to set or unset
XVT_COLOR_ACTION action) // set or unset the colors
```

xvt_ctl_set_colors sets or unsets the colors for a single control. This function overrides any color values you set previously for the control, but only for the XVT_COLOR_COMPONENT of the colors array. All other colors used by the specified control are not affected. To set the default colors for a control, use NULL for the value of colors. An action value of XVT_COLOR_ACTION_SET sets the control colors for the color components specified in the colors parameter. An action value of XVT_COLOR_ACTION_UNSET sets the control colors for the color components specified in the colors parameter to colors inherited from the control's container, the colors owned by the application, or the system default.

xvt_win_set_ctl_colors sets or unsets the colors for all existing controls in window win and all controls that you create after setting the colors. It will not change the colors of controls in other windows. This function overrides any color values you set previously for the controls in the window, but only for the XVT_COLOR_COMPONENT of the colors array. All other colors used by the window's control are not affected.

Note: For controls with color components set individually, the components that were set *will not* be affected by this color change. The components that *were not* set *will* be affected. For example, if a pushbutton has blue set for the foreground color and the window has red set for the background color, the background of the pushbutton will be red.

To set the default colors for controls in a window, use NULL for the value of colors. XVT_COLOR_ACTION_SET and XVT_COLOR_ACTION_UNSET work as described above. Note that this function does not affect the colors of the container decorations or any other colors that appear in the container itself.

The following Portability Toolkit functions allow you to get the currently-defined color settings:

```
XVT_COLOR_COMPONENT *xvt_ctl_get_colors( WINDOW ctl_win)
```

and

```
XVT_COLOR_COMPONENT *xvt_win_get_ctl_colors( WINDOW win)
```

Q: *Where are all new features of the PTK documented?*

A: New functionality is outlined in the *XVT Portability Toolkit Reference* and in the *XVT Portability Toolkit Guide*, both of which you will find in the documentation. XVT has chosen to use an electronic format to make reference information clearer, easier to find, and more usable.

In addition to documenting new functionality, the online *XVT Portability Toolkit Reference* contains sections on each of the following topics:

- XVT Portable Attributes
- XVT Events
- XVT Data Types
- XVT Constants
- XVT Functions
- URL Statements
- Help File Statements
- Tools

Q: *How do standard fonts map to multibyte fonts?*

A: XVT does not automatically map to multibyte fonts. In order for your application to use multibyte fonts, you must first Internationalize and Localize your application, using the methods detailed in Chapter 19 of the *XVT Portability Toolkit Guide*. You must also install the multibyte fonts appropriate for the language you intend to use, according to your system guidelines. This will allow the fonts to be available to your XVT application.

Presumably, you will be translating your application to one or more languages. If you have properly internationalized your application, all your text and font references exist only in your resource file. When you translate your text, you should also setup the font and font_map resource approriate for each language.

To set a multibyte font, you must modify the URL font or font_map statements of your application to contain native fonts appropriate for the language.

XVT supplies the following LANG_* xrc compiler options (files in your **ptk/include** directory):

- LANG_JPN_SJIS supports Japanese in Shift-JIS code (file **ujapsjis.h**)
- LANG_GER_IS1 supports German in ISO Latin 1
- LANG_GER_W52 supports German in Windows 1252
- Files for English, French, and Italian are also provided

These options and others are listed and discussed further in the *XVT Portability Toolkit Guide* and the *Guide to XVT Development Solution fo C++*.

XVT cannot guarantee which character set your customers will use. There is more than one set available for many languages. Because the font to which you map must be available on your customer's system in order for your application to run, a survey of your proposed customer base may be in order.

The availability of these fonts and other system setup issues should become part of the installation requirements for your application, or the fonts should be installed with your application.

Q: *I've completed development and thoroughly tested my application.I understand the XVT Portability Toolkit has compile time optimization. How do I enable it?*

A: In order to understand how XVT compile time optimization works, some knowledge of the XVT Portability Toolkit implementation is required. The XVT Portability Toolkit is implemented in two layers. The top API layer, the functions of which are listed in the *XVT Portability Toolkit Reference*, is called directly by your application. This layer performs error checking of all input parameters and sometimes other validation before calling the internal layer.It is the internal layer that contains the implementation of the functionality.

XVT provides a compile time symbol, XVT_OPT, which, when defined during application compilation, redefines the top level function names to directly call the internal API functions through macros. This bypasses the parameter checking provided by the top layer and eliminates an extra stack level for each XVT API function. You can also leave XVT_OPT undefined, allowing for the specific optimization of your application code. The header file **xvt_opt.h** contains the macro definitions of the XVT API functions that are optimized.

The optimization will not eliminate all error checking from the XVT Portability Toolkit. Rather, it will eliminate only those errors related to XVT API function parameters. Also, because the top layer sets up the error frames for function information, any errors that do occur may have fictitious results for the function stack trace.

XVT recommends this option be used only after you have completed development and have thoroughly tested your application. Attempting to use this option too early in your development process may result in application crashes and other odd behavior due to improperly called functions that would otherwise have been checked and diagnosed by the top API layer.

Q: *I'm not sure I understand the M_* values for DRAW_MODE as stated in the XVT Portability Toolkit Reference. What exactly am I supposed to see?*

A: The following “Draw Mode Definitions” section shows the different drawing modes supported by XVT. There is also an explanation of what these modes will do if you are drawing in black or white on either a black or a white source pixel.

See Also: For more information, see “Draw_Mode” under “XVT Data Types” in the XVT Portability Toolkit Reference.

Note: On systems that use a 256-color palette, and not 24 bit color, information in the charts will hold true only for black and white because the palette indices are used for ORing and XORing, not the color values themselves. Because there are no definitive (or at least portable) rules about what color is held in a given index, there are absolutely no guarantees as to what your results will be. 129 xor 1 will always be 128, but index 129 might be yellow, 1 might be white, and 128 might be off-puce. The application can attempt to force a palette, but the colors present will be a random mix based on what applications are currently running and what applications have run in the past in the same session.

The following code and draw mode definitions demonstrate the problem more clearly:

```
typedef enum {   /* drawing (transfer) modes */
        M_COPY,
        M_OR,
        M_XOR,
        M_CLEAR,
        M_NOT_COPY,
        M_NOT_OR,
        M_NOT_XOR,
        M_NOT_CLEAR
} DRAW_MODE;
```

Draw Mode Definitions

M_COPY:

- If you draw black, source pixel will be forced to black
- If you draw white, source pixel will be forced to white

M_OR:

- If you draw black, source pixel will be forced to black
- If you draw white, source pixel will be left as is

M_XOR:

- If you draw black, source pixel will be inverted
- If you draw white, source pixel will be left as is

M_CLEAR:

- If you draw black, source pixel will be forced to white
- If you draw white, source pixel will be left as is

M_NOT_OR:

- If you draw black, source pixel will be left as is
- If you draw white, source pixel will be forced to black.

M_NOT_CLEAR:

- If you draw black, source pixel will be left as is
- If you draw white, source pixel will be forced to white

M_NOT_COPY:

- If you draw black, source pixel will be forced to white
- If you draw white, source pixel will be forced to black

M_NOT_XOR:

- If you draw black, source pixel will be left as is
- If you draw white, source pixel will be inverted

Q: *What is the difference between NText and CText?*

A: NText and CText each display a single line of text and provide alignment options within their frames. Although their basic functions are similar, each class has unique characteristics that make it better than the other in different situations.

The NText class is derived from the CNativeView class. Native views have the look-and-feel of objects provided by the native window manager. They look slightly different from platform to platform. Visually and functionally they fit in with the analogous graphical items on the target platform. They are not implemented by XVT-DSC++, but by native toolkits, so you have less flexibility in manipulating them. Native views don't know how to print themselves. Since native views are derived from CView, they have all of the capabilities of other objects at the view level. As a native view, NText is defined by platform-specific resources. For example, it uses the system font and color as defined by the window manager.

You should use NText when you want your application, or parts of your application (certain dialog boxes, for example), to have the look-and-feel of objects created by the native window manager.

The CText class is derived from the CView class. Unlike NText, which is drawn by the native window manager, CText creates drawn text which looks the same across all platforms. It allows user and program control over its font properties and colors. For example, it allows you to choose from a variety of font families (Times, Helvetica) and styles (italics, boldface). It can dynamically change its size as its contents change. It can change its placement and alignment at runtime. It can also output itself to a printer.

You should use CText when you want more creative control over the appearance of your text, when you want your text to appear the same across all platforms, or when you want to give the user creative control over the appearance of text in your application.

See Also: For more information, see “CText” and “NText” in the *XVT DSC++ Reference* and also look for references to CText and NText in the *Guide to XVT Development Solution for C++*.

The “Textual Views” chapter in *Introduction to C++ for Developers* is also helpful.

Q: *Is there a way to implement zooming in DSC++?*

A: The following solution does not use CUnits and will result in correctly updated wireframes, scrolling, sizing, dragging, and so on.

Create a new class called ViewInfo, for example. The purpose of ViewInfo is to keep track of the location where the view was created. Each time that a new view is inserted in the CScroller, create an associated ViewInfo. Fill the associated ViewInfo with the view's creation-frame and a pointer to this view. This ViewInfo instance is then appended to a RWOrdered.

When the zoom factor changes, for example, to 150%, iterate through the RWOrdered, and tell the view, which is pointed to size 1.5 times its original frame. Once all views have processed, call xvt_dwin_invalidate_rect on the CScroller. Everything should successfully redraw. If a CWireFrame has been moved, it generates a WFSizeCmd, and the DoCommand looks up in the RWOrdered to update the creation coordinates according to the actual zooming factor.

The following code illustrates:

```
class ViewInfo : public RWCollectable {
public:
        ViewInfo( CView* theView, const CRect& theRect ) ; ~ViewInfo( ) ;
        virtual CRect& SetFrame( const CRect& theRect ) ;
        virtual CRect GetFrame( void ) ;
        virtual CView* GetView( void ) ;
protected:
        CRect itsCreationFrame; CView* itsView;
private: }  ;
```

A fundamental problem is equating the Size() method with zooming. Here are the issues:

- What happens when a view is resized in the usual way? For example, as a pane in a splitter window, a subview may be resized to be twice as wide. Is this equivalent to zooming by 200%?
- What happens when a view is moved in the usual way? Will the associated ViewInfo object need to refresh itsCreationFrame? How would this be done?
- What happens when a native control is zoomed? For example, if a NListButton is told to zoom (resize), the edit box will remain the same height.

- What happens when a CPicture (or a CPictureButton, etc.) is told to resize? Will the picture stay intact?
- What happens to subviews within subviews? The splitter will be resized, but the oval will stay the same.

It should be possible to resolve all of these issues without the need to subclass everything. Expand on what has been started in the ViewInfo class above, and envision a type of visitor attached to the switchboard called a CZoomHandler.

A CZoomHandler will have a zooming factor attribute. If this is set to 100%, it will not do anything. A CZoomHandler will intercept E_ UPDATE events at the Switchboard and perform a deep traversal through the window's object heirarchy, via DoDraw(). The CZoomHandler will render each view as it sees fit: On some views, it may just temporarily reset its size attributes and then call its Draw() method. On others, it may do its own drawing to handle some of the tougher issues listed above.

Q: *How do you create global variables for use in a DSC++ application?*

A: The best way to use variables that can be accessed globally from your application is to use them in a real global object, such as the CApplication- derived object. The application object should encapsulate the variables and make them accessible only through member functions. For instance, if the application object has a private variable named theVariable, then the application object might have a member function named SetTheVariable() and another called GetTheVariable(). This approach is a standard mode of operation in most object-oriented applications.

Some prefer to use the CGlobalUser class. This class, however, does not encapsulate and protect data as well as using a more object-oriented approach as described above. In case you choose to use the CGlobalUser class, the following paragraphs describe how.

The CGlobalUser class object has application global scope and can be used to access any global variables you may need. You can find documentation for this class in the *XVT-DSC++ Reference*.

The CGlobalUser class is utilized as follows:

1. Copy the **CGLBLUSR.H** file from the **pwr/include** directory to your development directory. You should rename the original file so that the compiler will see your own copy.

2. Add public class variables to your copy of the header file as follows:

```
//////////////////////
//Add items as needed// //////////////////////

class CGlobalUser : public CNotifier {
        public:
                CGlobalUser(void) {}
                XVT_HELP_INFO xd_help_
                info; FILE_SPEC* initFile;
                SECURITY_LEVEL userLevel;
};
```

3. In your application's startup member function, create an instance of CGlobalUser and pass it to CBoss: IBoss as follows:

```
//////////////////////////////
// Call IBoss to instantiate //
// the CGlobalUser object    // //////////////////////////////

void CDEMOApp:
:StartUp();
{
        CApplication::StartUp(); IBoss(new CGlobalUser); DoNew();
}
```

4. Access the global variables through the CBoss's GU pointer, as follows:

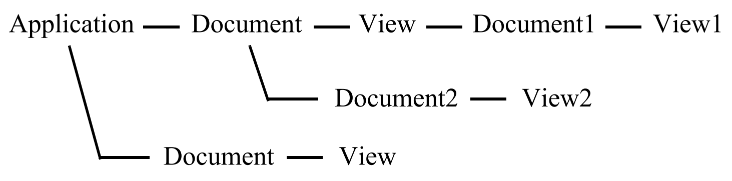
```
...
// Access the global userLevel GU->userLevel = SUPER_USER; ...
```

5. Destroy the GU pointer in the application's ShutDown member function, as follows:

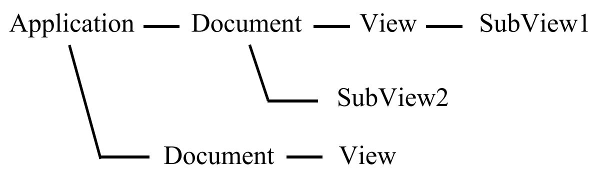
```
///////////////////////////////////
// Destroy GU and set it to NULL  // ///////////////////////////////////

void CDEMOApp:
:ShutDown(void)
{
        delete GU;
        GU = NULL;
        CApplication:
:ShutDown();
}
```

Q: *In the Application-Document-View hierarchy, can I have more levels of Document-View? For example, can I have a hierarchy like the following:*



In other words, if there are only three levels in the hierarchy, I have to put all data access/management code in one document and then use this single document to maintain all its views, as illustrated below?



A: No, you cannot have multiple levels of documents using the DSC++ framework. CDocument objects must be parented to one CApplicaton instance, just as CWindow objects are parented to a single CDocument instance. However, this arrangement gives you plenty of power for managing document data.

It might help to make a distinction between two different concepts that are used in the DSC++ framework. One is the "Application-Document-View" concept, and another is known as the "Model-View-Controller" design pattern. These two patterns can be used separately or together to build your application's data-flow structure.

It is true that you have only one level of views that are windows into the data in a document. However, it makes sense that there is only one level of complexity in this model. The real purpose of the App-Doc-View idea is to help the developer visualize which windows are looking at which separate groups of data.

In the App-Doc-View paradigm, it is the document's role to be the conduit of data flow between the data level and the presentation level of a two- or three-tier architecture. A document represents, in all its complexity, an entire, independent data set. Even if your presentation draws its data from several different sources, it can still be thought of as one data set, managed by a single document.

The App-Doc-View concept helps in laying out applications that have many windows that look into one data set, and a separate collection of windows that look into an entirely different data set. In your case, you may not have this type of complexity.More complex documents probably should be broken up into more manageable models, where the document manages (creates and destroys) these models. Each model is designed to solve one piece of the overall project.

In some cases, you may have a single window that looks into two separate data sets. In such a situation, the “Model-View-Controller” design pattern will be more appropriate. This design is borrowed from the Smalltalk programming environment to help keep all the windows into a data set in sync so they all have the same data at the same time.

An MVC object structure can be as complicated as needed. When the state of one model changes, all other dependent models may be automatically notified and updated via the controllers to which the models are registered.

You can implement this with a document that owns many data models (use the CModel class). Each model has a controller that decides whether windows can change or read the data. You register each of the views with the data controller (use CController). These views can be implemented as CViews, CSubviews, or CWindows. When the data in the model changes, the controller will send a message to the appropriate views so that they can update themselves with the data. The document would manage both the data models and the views themselves.

In Architect, you can visualize the layout with the Application-Document-View graph. However there is no visual way to represent the Model-View-Controller idea in Architect because this design pattern has less to do with the layout of the application, and more to do with the internal data structures.

These two separate concepts have their own unique uses as generic design patterns. Thinking about object-oriented programs in terms of abstract design patterns has proven quite useful to many object-oriented programmers. A good book on the topic is *Design Patterns:*

Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson, and Vlissides.

Q: *How do you print hidden views, multiple pages, or native controls in DSC++?*

A: The default behavior of Architect-generated code is to print the contents of a window on a single print page. DSC++ has built-in functionality for printing the screen images of most drawn or rendered objects, including CSubView-derived objects. Anything which lies beyond the boundaries of the window is clipped in the printed output. There is no functionality in DSC++ or the C-level Portability Toolkit (PTK) for printing images of native controls, specifically anything that inherits from the class CNativeView.If your application only needs to be able to produce simple screen shots of custom views drawn on a single page, you probably do not need to override any printing methods.

However, many applications need to be able to print text or graphics on multiple pages. Others may need to print portions of the view that demonstrate how to print the contents of a text editor object on multiple pages.

To understand Printing in DSC++, consider that there may be several overloaded versions of DoPrint acting in a single print process. CPrintManager has the DoPrint member function, CDocument has DoPrint, and CView also has DoPrint. These three implementations coexist, and they do different tasks. If you look at the DSC++ hierarchy, you can see they cannot override one another.

Printing is started when the user selects the Print option from the File menu. This generates the standard M_FILE_PRINT menu command. The menu command goes to your window's DoMenuCommand method. It propagates up from there to the default CWindow menu, then the default CDocument menu command. CDocument will then call the CDocument-derived DoPrint Method.

If your application overrides the CView::DoPrint, note that your overridden DoPrint function will not get called with the default Architect code. This is because the CView class does not inherit from CDocument, and it is the CDocument DoPrint that gets called by default. You will need to add your own code to call the proper print method. Usually this code is added at the window object level.

At the document level, the DoPrint method inserts each of the document's windows as an entry (or page) in the print queue, and calls the CPrintManager::DoPrint. The CPrintManager's DoPrint starts a PTK print thread. If you override CView::DoPrint, your function will also call CPrintManager::DoPrint.

The PrintThread function looks at every item in the print queue and opens a print page for each one. It calls an item's DoPrintDraw and then closes the page. This way each view in the queue gets exactly one page.

The default DoPrintDraw, generally at the CView level, simply sets the output device to be the printer, prepares the clipping region of the view, and calls the view's PrintDraw. PrintDraw is not called if the view is invisible. PrintDraw then does the drawing to the print page. In many cases, PrintDraw just calls Draw, the same routine that draws to the screen. Drawing to the screen or to the print page works interchangeably, depending on how the output device is set.

The secret to printing multiple pages it to override the DoPrint method that inserts pages into the print queue. Every time the CPrintManager's Insert method is called results in a page of printed output.If you want a view to appear on several pages, call Insert once for each page with the same view as its parameter. If the objects to be printed are within a virtual frame, you can scroll hidden views into the visible portion of the frame before you enter the views in the print queue.

The overridden DoPrint also needs to figure out how many pages a view will occupy. Often times, this requires converting from printer dot units to pixel units. For this, you need to create a units object with dynamic mapping for your application.

Q: *Why do my deserialized fonts print too small?*

A: Font serialization is a means by which the specifications for a logical font file can be saved to a file for later use in the program or in another program. When a font is serialized the following information is extracted from the font: its family, size, style, and native description.

The fact that the native descriptor is stored can cause a problem. When a font is deserialized and it contains the native descriptor, XVT will no longer map the font. This is by design because it is possible to customize the native font description beyond the access functions provided by XVT.

A symptom of this problem is when a font is deserialized and everything looks OK in the window, but then the same font, when used to print, results in very small text.

The solution to the problem is not to store the native descriptor in the serialized buffer. A typical serialized font looks as follows:

```
01\system\0\12\WIN01/-16/0/0/0/400 /0/0/0/1/0/0/2/32/System
```

Everything from the WIN01 on is part of the native descriptor. A serialized font without a native descriptor looks as follows:

```
01\system\0\12\
```

This version will continue to be mapped as necessary after deserialization. The following is the code to remove the native descriptor from the serialized font:

```
FILE *fp = fopen("font.ser", "w+"); xvt_font_unmap(newFontInfo.itsFontID);
xvt_font_set_native_desc(newFontInfo.itsFontID," "); save_font_to_file(fp,
newFontInfo.itsFontID); xvt_font_map(newFontInfo.itsFontID,xdWindow);
fclose(fp);
```

And following is the actual font serialization code:

```
void save_font_to_file(FILE *fp, XVT_FNTID font_id) {
  char *buffer; long length;
  length = xvt_font_serialize(font_id,NULL,0);
  buffer = (char *)xvt_mem_alloc(length); if(buffer)
  {
    if(xvt_font_serialize(font_id,buffer,length))
    {
      fputs(buffer,fp);
      xvt_dm_post_note("Serial buffer: %s", buffer); }
    else
      xvt_dm_post_warning(
                  "Could not serialize font"); xvt_mem_free(buffer);
  }
  else
    xvt_dm_post_warning(
                  "Could not allocate memory for buffer"); }
```

Finally, the following is an example of font deserialization code:

```
/* this function reads a font from a file. */
void read_font_from_file(FILE *fp,XVT_FNTID font_id) {
  char buffer[BUF_SIZE]; fgets(buffer,BUF_SIZE,fp);
  xvt_dm_post_note("Buffer read from disk: %s", buffer);
  if(xvt_font_deserialize(font_id,buffer)==FALSE) xvt_dm_post_warning(
                  "Could not deserialize font"); }
```

Q: *Is there a good way to track event processing?*

A: Yes. A good way to track event processing is to use the functions xvt_debug and xvt_debug_printf from the XVT Portability Toolkit. These functions, when used from within an event's switch statement, allow the programmer to print lines of text to a file with some explanation of which event is taking place. These functions do not cause events themselves, so they are ideal for tracking event flow within an application. The functions use regular sprintf arguments (passes a char*). Strings are passed to a file named **debug** in the working directory. Alternately, a filename can be specified by setting ATTR_ DEBUG_FILENAME.

Consider the following code, for example:

```
{
if (xdEvent->v.active)
{      xvt_debug_printf("Focus has entered window"); /*
Window has gained focus
}
else { xvt_debug_printf("Focus has left window"); /*
       Window has lost focus */
}
```

This code, when placed within an e_focus for a window, will place the appropriate message into the file debug, alerting the programmer to whether a focus event has been sent to the window.

There is a slight difference in the use of xvt_debug and xvt_debug_printf. xvt_debug is conditional upon the following two things:

- A preprocessor command, #define DEBUG must be *before* #include "xvt.h"
- The debug file is present in the working directory at run-time

Q: *How can I implement keyboard navigation using DSC?*

DSC makes it simple to implement keyboard navigation for all of your controls within a window. Design can generate the code for the programmer, or the programmer can implement the feature through calls to the PTK.

To implement navigation in Design, do the following:

1. When you make a new window, go into the attributes and check the Keyboard Navigation checkbox.

2. Set the creation order by giving the window focus and then setting the Creation Order under the EDIT menu on the menu bar.
3. Generate your project.

To implement through coding function calls, do the following:

1. In the E_CREATE event for your window, add the following code:

   ```
   xvt_nav_create(xdWindow, NULL);
   ```

   The first parameter, xdWindow is the window handle passed to the event handler, and the second parameter is a valid SLIST (string list). NULL will cause your window to use its immediate child windows in order of their creation.

2. In the E_DESTROY event for your window, add the following code:

   ```
   XVT_NAV nav = xvt_win_get_nav(xdWindow); if (nav) xvt_nav_destroy(nav);
   ```

   This code declares an XVT_NAV object called nav which stores the result of the xvt_win_get_nav function. xvt_win_get_nav passes the window handle as its parameter and returns a valid XVT_NAV object if one exists, or NULL if one doesn't exist. If a valid XVT_NAV object is returned, the if statement invokes xvt_nav_destroy, which destroys the navigation object.

Note: XVT will destroy the navigator when a window is destroyed but we suggest that you make the call for create/delete bounding.

Additional Notes:

1. A window must be of type W_DOC, W_PLAIN, W_DBL, W_MODAL, or W_NO_BORDER in order to use Keyboard Navigation.
2. A control or child window can be added to an existing XVT_NAV object through the use of xvt_nav_add_win, as follows:

   ```
   BOOLEAN xvt_nav_add_win(XVT_NAV nav, WINDOW win,
               WINDOW refwin, XVT_NAV_INSERTION where);
   ```

   where XVT_NAV is the navigation object, WINDOW win is the child window or object to be added , WINDOW refwin is the child window or object to used as the reference for insertion, and XVT_NAV_INSERTION where is a constant of type XVT_NAV_POS_BEFORE, XVT_NAV_POS_AFTER, XVT_NAV_POS_FIRST, or XVT_NAV_POS_LAST (This parameter takes the place of using the Creation Order feature in Design.).

This function returns TRUE if the object was placed in the navigation order, or FALSE if the function did not succeed.

Q: *How do I create list boxes with tab stops?*

A: The LBS_USETABSTOPS flag needs to be set for the control. Because this flag must be set before creation, it cannot be set with SetWindowLong(). LBS_USETABSTOPS does work if it's set in the ATTR_WIN_CREATEWINDOW_HOOK function, as follows, for example:

```
static long task_eh XVT_CC_ARGS((WINDOW xdWindow,
          EVENT *xdEvent));

#include "windows.h" void CWHook(DWORD* es, LPCSTR*
          cn, LPCSTR* wn, DWORD* st, int* x, int* y, int* w,
          int* h, HWND* p, HMENU* m, HANDLE* i, void** cp) {

     if (!strcmp(
               *cn, "Listbox") && *m == WIN_101_LBOX_2) {

          *st |= LBS_USETABSTOPS; } }
```

This will set the LBS_USETABSTOPS for the listbox with ID of WIN_101_LBOX_2.

Q: *When I change the background/foreground colors of some controls, the change doesn't show up in the compiled application. Why not?*

A: There are three possibile explanations for this scenario:

1. Developing or running the application on Windows 95 can cause this problem to occur because, unfortunately, Windows 95 does not support modifying the background/foreground colors of PushButtons.
2. The problem may also be caused by the fact that the feature is not supported by MS Windows CTL3D libraries (for example, modifying the background/foreground color of the actual button on a RadioButton, or the checkable box on a CheckBox).
3. Another cause could be that while Win32/64/64 does support the feature, on some platforms, the CTL3D library must be included to see the effect. Modifying the background/ foreground color of a pushbutton requires the user to include the

MS Windows CTL3D library. To include the CTL3D library add the following line before xvt_app_create:

```
xvt_vobj_set_attr(NULL_WIN,ATTR_WIN_USE_CTL3D,TRUE);
```

Note: Design uses the CTL3D library. This means that TestMode in Design will display the modified control colors even though the required attribute has not been set in the code for the compiled application. One possibility for users is to include the above line in the **DESIGN.CFT** file so that by default it will be included in every project. For more information, see "Control Component Colors" in Chapter 8 of the *XVT Portability Toolkit Guide*.

Q: *How do I put XPO Toolbars/Statusbars in an MDI Task Window?*

A: The implementation of these objects is simple. The user needs only to change a flag to get the objects to appear in the task window. Use the following steps:

1. In Design, create a window to lay out the toolbar and statusbar as you would with any other control. This window is only used to lay out these objects. When you double click on the toolbar and statusbar, the attribute TASK_WIN is listed with the other attributes. Change this attribute to TRUE.
2. To turn on MDI mode, modify the Application Main Code tag as follows:

```
#if (XVTWS == Win32/64/64WS)

xvt_vobj_set_attr(
            NULL_WIN, ATTR_WIN_MDI, (long) TRUE);

#endif

xvt_app_create();
```

3. You must instantiate the window that was used to layout the toolbar and statusbar. The user can allow this window to be invisible and remain for the duration of the application, or he or she can call xvt_vobj_destory(xdWindow) during the E_CREATE of that window.

When a toolbar is placed in the task window, any notices it generates are sent to the E_USER of the Application (TASK_WIN), not the window in which you placed the toolbar when using Design.

Q: *How do I include native headers on NT?*

A: Although including native headers is discussed in section 3.1.1 of this book, there are a couple of things specific to using DSC++ that need to be taken into consideration.

First, isolate the native SDK function calls to a single source module, if possible. This will aid in code maintenance and clarity. Enter #define XVT_INCL_NATIVE at the very top of the source module. Defining this will trigger the necessary steps in **xvt_plat.h** located in the **..\ptk\include** directory. It is imperative that XVT_INCL_NATIVE is defined before **xvt.h** is included.

The top of the source module should resemble the following:

```
#define XVT_INCL_NATIVE #include "xvt.h"
#include "winsock.h" //or other native headers
#ifdef CreateWindow
#undef CreateWindow
#endif
#ifdef FindWindow
#undef FindWindow
#endif

#include "AppDef.h
.
.
. // other DSC++ includes
```

Because they are used as macros in the native header files, and inside of DSC++ they are methods, CreateWindow and FindWindow must be undefined. The compiler will give errors if they are not undefined.

If there is a need to have native information in a DSC++ header file, such as a member variable of a native type, then declare the variable in the DSC++ header file in the same way that it is declared in the native header file. Following this procedure is especially important if you are using Architect. When native information is placed in a header file with the aid of XVT_INCL_NATIVE, compilation of the factory files will fail. This is because the order of includes in the factory source modules cannot be controlled by the user, and **xvt.h** will be included prior to the file containing XVT_INCL_NATIVE.

Q: *How can derived classes be traced with a debugger into the application framework?*

A: DSC++ is not shipped with a debug version of the application framework. While the provided makefiles could be used to build the framework, the resulting library is virtually unusable with the Microsoft Visual Studio debugger.

The application framework may be selectively built for debugging should it be necessary to trace either derived or base classes. To achieve this, determine how far back into the hiearchy the code needs to be traced for a given class and add the base source files into the application's make file.

For example, to trace the CPassword class in the DSC++ example set (available for download at the FTP site) back through CView:

Classname:	**Source File:**
CPassword	**cpasswrd.cpp**
NLineText	**nlinetxt.cpp**
CNativeTextEdit	**cntvtxte.cpp**
CView	**cview.cpp**

1. Set the required compiler and linkage options for debugging into the makefile. (These are compiler specific. Consult the appropriate documentation.)
2. Add **cpasswrd.cpp**, **nlinetxt.cpp**, **cntvtxte.cpp**, and **cview.cpp** into the project makefile. The source modules may be copied into the project's directory or left in **...\pwr\src** as desired. (While most of the eight-dot-three filenames are self evident abbreviations of their class names, the **PWRDEF.H** file contains the **#DEFINES** used for the naming conversion.)
3. Build the project.

The application may now be run with the debugger. If a breakpoint is set inside of CPassword, the code execution may now be followed into any of the above base classes.